The Thesis committee for Hayes Elliott Converse
Certifies that this is the approved version of the following thesis:

Non-Semantics-Preserving Transformations For Higher-Coverage Test Generation Using Symbolic Execution

APPROVED BY
SUPERVISING COMMITTEE:

Sarfraz Khurshid, Supervisor

Dewayne Perry

Non-Semantics-Preserving Transformations For Higher-Coverage Test Generation Using Symbolic Execution

by

Hayes Elliott Converse, B.S.

Thesis

Presented to the Faculty of the Graduate school
at the University of Texas at Austin
in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science in Engineering

The University of Texas At Austin
May 2016

Acknowledgements

The initial conception and driving force behind the ideas presented herein can be credited to Sarfraz Khurshid, professor of computer engineering at UT Austin. He similarly enabled the research that inspired this study, Shiyu Dong's study of compiler optimizations and their interplay with symbolic execution. The author would also like to thank Shiyu for enthusiastic co-operation.

This work would not have been possible without the constant support and advice of Oswaldo Olivo of UT Austin's Computer Science department; each meeting with him opened up doors that were previously closed.

The author must extend the most heartfelt thanks to his longtime friend Laura Jones for her willingness to spend many hours editing a paper in a subject she knew nothing about.

Finally, the author would like to thank Mark Harman and Christian Cadar for their contributions to the foundational knowledge base of this project.

This work would funded in part by the National Science Foundation (NSF Grant Nos. CCF-0845628 and CNS-1319858).

Non-Semantics-Preserving Transformations For Higher-Coverage Test Generation Using Symbolic Execution

by

Hayes Elliott Converse, M.S.E.

The University of Texas at Austin, 2016

SUPERVISOR: Sarfraz Khurshid

Symbolic execution is a well-studied method that can produce high-quality test suites for programs. However, scaling it to real-world applications is a significant challenge, as it depends on the expensive process of solving constraints on program inputs. Our insight is that non-semantics-preserving program transformations can reduce the cost of symbolic execution and the tests generated for the transformed programs can still serve as quality suites for the original program. We present several such transformations that are designed to improve test input generation and/or provide faster symbolic execution. We evaluated these optimizations using a suite of small examples and a substantial subset of Unix's Coreutils. In more than 50% of cases, our approach reduces the test generation time and increases the code coverage.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Symbolic execution [1, 2] is a powerful method for software verification and validation, with applications in the processes of automatic mutation generation, data flow testing, and patch generation, among others [3–5]. A symbolic execution engine performs a dynamic path-based analysis of a program, allowing it to explore a large percentage of that program's bounded behavior space. Suitable test inputs are selected using the solutions to each branch's path condition generated by an SMT solver [6]. This allows the engine to generate large, high-quality test suites [1, 2, 7–13]. However, the behavior space of a program grows very quickly with its size and complexity in a phenomenon known as state space explosion. Subsequently, the time demands of the requisite SMT solver calls become burdensome to the point of in-feasibility. This makes symbolic execution impractical for many real-world applications [14].

This scaling problem has received a significant amount of attention from the software testing community. Traditionally, symbolic execution performs a time-limited depth-first search of a program's control-flow graph, which maximizes depth of coverage at the expense of thorough analysis of programs with many deep branches. Several techniques have been developed for improving symbolic execution times and thus allowing deeper exploration, including loop summarization [15], various heuristics [16], path analysis for intelligent selection [17, 18], parellelization [19, 20], memoization [21], and ranged analysis [22], among others. Recently, Dong et. al. [23] brought attention to the interactions between symbolic execution and standard compiler optimizations intended to improve the speed of execution of programs on concrete inputs. They found that these semantics-preserving optimizations can decrease the speed of symbolic execution, especially when applied in combination. Symbolic execution must query an SMT solver at each junction in the program's execution, and traditional compiler optimizations can make it more difficult by transforming variables or otherwise adding clauses to the formula under consideration. This implies that symbolic execution is in need of a new class of optimizations, designed under a different set of assumptions.

In an attempt to make symbolic execution faster while preserving its advantages, we present a new series of non-semantics-preserving *testability* transformations [24] that are applied to programs before symbolic execution is performed. Symbolic execution of the transformed programs produces a test suite, which can then be run against the original program. The principle behind this approach is simple: given a program p and another program q with the same method signature, tests written for p can be executed against q. Our key insight is that if q's logic is simpler than p's, using symbolic execution to generate tests for q may be less costly than generating tests for p, and the code coverage on p using tests generated for q may still be similar to the coverage using tests generated for p. Non-semantics-preserving transformations can also be used to guide symbolic execution towards areas of the program that need particularly intensive testing. We use KLEE [7] as our symbolic execution engine in this paper, as its foundation on LLVM allows us to easily create and use new compiler optimizations.

We present our study of several new testability transformations designed end-to-end to improve symbolic execution in its traditional context (i.e., using time-limited depth-first search without caching). We equivalently refer to these transforms as "optimizations" throughout this paper, as their goal is to produce a program optimized for symbolic execution. We evaluated these optimizations on a well-studied group of programs, Unix's Coreutils, as well as several small examples as proof of concept. We also conducted the same experiments with caching enabled, as caching similarly decreases the number of calls KLEE makes to the SMT solver. Our initial findings show that these transformations can in fact increase code coverage and reduce symbolic execution times. Our findings are summarized below:

- Test suite generation can be improved using non-semantics-preserving transformations, providing 100% to 418% of the coverage of the original test suite, with an average of 127%.
- In cases where transformed programs performed worse than their original counterparts, they did so with variable loss of accuracy, covering anywhere from 23.47% to 97.7% of the code covered by the original test suite, averaging about 75%.

- Enabling caching produces a small but positive change in performance, including small increases in line coverage and decreases in both execution time and the number of SMT solver queries.

This thesis makes the following technical contributions:

- The design and implementation of the first non-semantics-preserving testability transformations for symbolic execution.
- Evaluation of these transformations on a significant group of programs.

Our work explores only a small space of the possible non-semantics-preserving transformations that can help scale symbolic execution. We hope that our work will motivate further research into the possibilities offered by this promising development, as symbolic execution is an incredibly powerful method with great potential in the field of systematic testing and verification.

Chapter 2

Foundations

This section provides a brief overview of the central concepts behind symbolic execution, LLVM and its symbolic execution engine, KLEE, and non-semantics-preserving transformations.

2.1 Symbolic Execution

Symbolic execution treats program inputs as variable rather than concrete. At each control point (or *branch*) in the program's execution, these symbolic inputs are combined into a Boolean formula representing the necessary conditions for the program to reach that point. These formulas, known as *path conditions*, naturally grow more and more complex with each subsequent control point along a path, so solving them similarly becomes a more and more time-consuming task. This presents a problem, as each of these formulas must be solved to ensure that a given path is feasible. If it is, the solutions to that formula can become a test input for that path. As noted above, there have been a number of approaches developed with the goal of addressing this bottleneck, both in the fields of SMT solving and symbolic execution. However, present approaches do their best not to change the *semantics*, or behavior, of the program under test.

2.2 LLVM and KLEE

The LLVM framework [25, 26] is a powerful group of compilation and execution tools. Chief among them is the LLVM core, which allows programs to be compiled, represented, and manipulated through the LLVM Intermediate Representation (IR). This independent back end allows optimizations to be developed and applied regardless of a program's front end. Additionally, this project provides the basis for KLEE [27], a symbolic execution engine that uses a user-specified constraint solver to identify the necessary input conditions for each path through a program's control flow, as discussed above. KLEE also supports user definition of bounds on the program's input space, complicating the path conditions but ensuring that the created tests are useful in the context of the program's desired application.

KLEE can produce test suites that cover 90% of the Coreutils on average [7, 28], given enough time. However, as full symbolic execution of non-trivial programs can take many hours, KLEE is usually run with a time bound. Given these conditions, improving the engine's performance is a matter of either making the process of SMT solving more efficient to allow more time for exploration and solver queries within the allotted time, or pruning infeasible, redundant, or otherwise inapplicable paths.

2.3 Compiler Optimizations and Testability Transformations

Standard compiler optimizations are *semantics-preserving program transformations,* i.e. the transformed program will provide the same output for a given input as the non-transformed program. This is a necessary precondition for any optimization that is intended to speed up a program in a deployment (i.e., concrete execution) setting. A compiler optimization would be of little use if it changed the program's behavior, regardless of increases to performance. A *non-semantics-preserving transformation,* conversely, is under no such obligation. The transformed program need not behave at all similarly to its progenitor. Clearly any such transformation would be useless as a compiler optimization meant for deployed programs. However, before software can be deployed it must be tested, and it is on this point in the development life cycle that our research is focused.

The transformations that we have developed fall under the heading of *testability transformations,* [24] which are used in the testing phase to enable some test data generation method, in this case, symbolic execution. A testability transformation produces an altered program that is in some way more suited to be used by the selected test data generation engine; it can be discarded after use and has no requirements with regards to semantics preservation. Testability transformations also permit alteration of the *test adequacy criteria*. KLEE's developers theorized about the possibility of using such transformations to improve symbolic execution [29], but this paper is the first to our knowledge to design and implement non-semantics-preserving testability transformations explicitly for this purpose. As our design goals in the symbolic execution setting are similar to those of compiler optimizations in the concrete execution setting, we have decided that it is appropriate to refer to our transformations as *testability optimizations* (shortened to *optimizations* for ease of discussion). These are distinct from traditional compiler optimizations, which we reference explicitly.

It is important to note that, in order to generate useful test cases, a few rules must still be obeyed when designing these transformations. Most importantly, the produced program must have the same method signature, as test cases take the form of program inputs. A mismatch between the transformed program and the original program in the number and/or type of inputs required would result in some or all of the produced tests being unusable for the original program. This also means that the transformed program must have the same return type as its source, although the return value can vary arbitrarily. Finally, the transformed program must still have an entrance point, so the first basic block must not be removed.

Chapter 3

Motivating Examples

This section provides several examples using small programs to demonstrate how programs can be optimized for symbolic execution using non-semantics-preserving transformations. The program in Figure 3.1 has two consecutive loops with identical loop conditions. Symbolic execution of this program using KLEE (with default settings) produces a test suite that provides 100% code coverage. Note that the control flow of the second loop is independent from the program inputs. Thus, removing the second loop, as in Figure 3.2, makes symbolic execution faster without decreasing the test suite's coverage. Control-flow graphs for both programs can be seen in Figures 3.3 and 3.4, respectively. Making this alteration by hand reduces KLEE's execution time from 10.9 hours to 23 minutes, providing a 28X speedup, and reduces the number of queries issued to the SMT solver from 15495 to 6823, i.e. a 2.3X reduction. This result is achievable using one of our transformations, described in Section 4.2.

The program in Figure 3.5 has a conditional structure inside of a loop. However, multiple symbolic executions of the loop are not necessary for complete coverage of the code within it. By adding an unconditional return statement at the end of the loop, as in Figure 3.6, we can significantly reduce the complexity of the structure while still providing 100% line coverage. The control-flow graphs for these programs (in Figures 3.7 and 3.8, respectively) show the change visually: the altered program's graph has no back edges. In this case, KLEE's execution time is reduced from 19.63 to 3.29 seconds, i.e., about a 6X reduction, and makes only 201 solver queries for the transformed program as opposed to the original's 1619, i.e., an 8X reduction. This is similarly achievable using one of our transformations.

```
 1 int main( int argc, char *argv[] ){
 2  int x, y, i, k = 0;
 3  x = atoi(argv[1]);
 4  y = atoi(argv[2]);
 5  for(i = 0; i < 10; i++){
 6   if(k < 20) {
 7    k += x*y; }
 8  }
 9  for(i = 0; i < 10; i++){
10   if(i < 5) {
11    k += x+y;  }
12  }
13  return k;
14 }
```

FIGURE 3.1: Program p with two loops.

```
 1 int main( int argc, char *argv[] ){
 2  int x, y, i, k = 0;
 3  x = atoi(argv[1]);
 4  y = atoi(argv[2]);
 5  for(i = 0; i < 10; i++){
 6   if(k < 20) {
 7    k += x*y; }
 8  }
 9  return k;
10 }
```

FIGURE 3.2: Program $p_{transform}$, equivalent to p with the second loop removed.

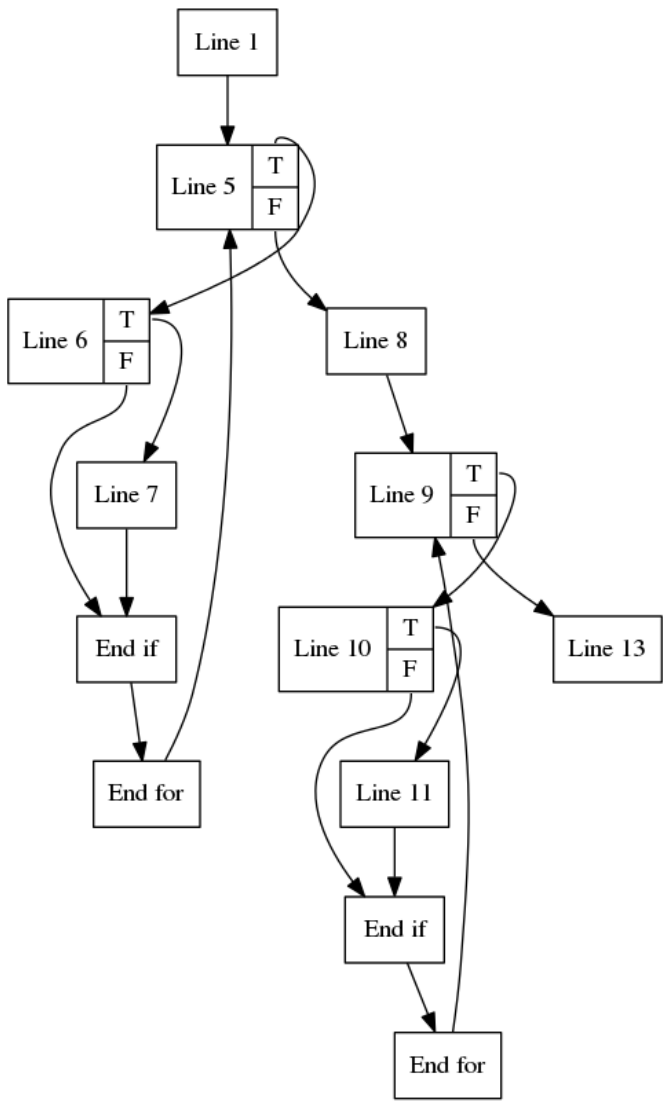

FIGURE 3.3: CFG for program p.

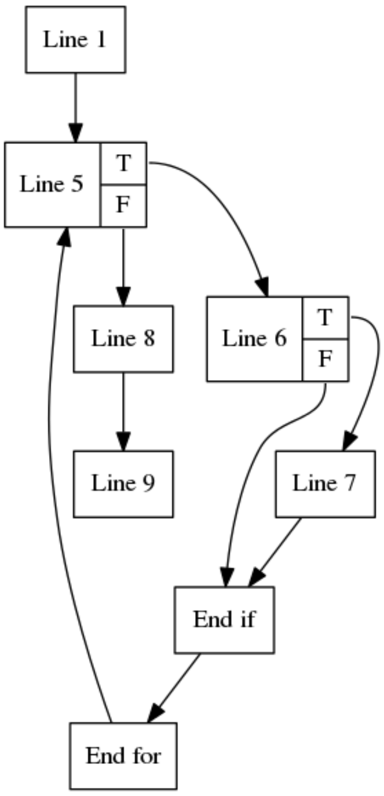

FIGURE 3.4: CFG for program $p_{transform}$.

```
int main(int argc, char *argv[]) {
 int a = atoi(argv[1]);
 int x = 0, y = 0, z = 0;
 if (a < 0){
  return -1;
 }
 while(a < 12){
  if(a >= 0 && a < 4){
   x++;
   a++;
  }else if(a >=4 && a < 8){
   y++;
   a++;
  }else if(a >= 8 && a < 12){
   z++;
   a++;
  }
 }
 return z;
}
```

FIGURE 3.5: Program q with a conditional structure contained in a loop.

```
int main(int argc, char *argv[]) {
 int a = atoi(argv[1]);
 int x = 0, y = 0, z = 0;
 if (a < 0){
  return -1;
 }
 while(a < 12){
  if(a >= 0 && a < 4){
   x++;
   a++;
  }else if(a >=4 && a < 8){
   y++;
   a++;
  }else if(a >= 8 && a < 12){
   z++;
   a++;
  }
  return 0;
 }
 return z;
}
```

FIGURE 3.6: Program $q_{transform}$, equivalent to q with the loop terminated after a single iteration.

These examples demonstrate that non-semantics-preserving transformations can preserve the quality of the test suites generated through symbolic execution. In the course of our experiments, we discovered that the transformations we devised can actually improve the quality of these test suites. Our specific results are discussed in Section 5.5.

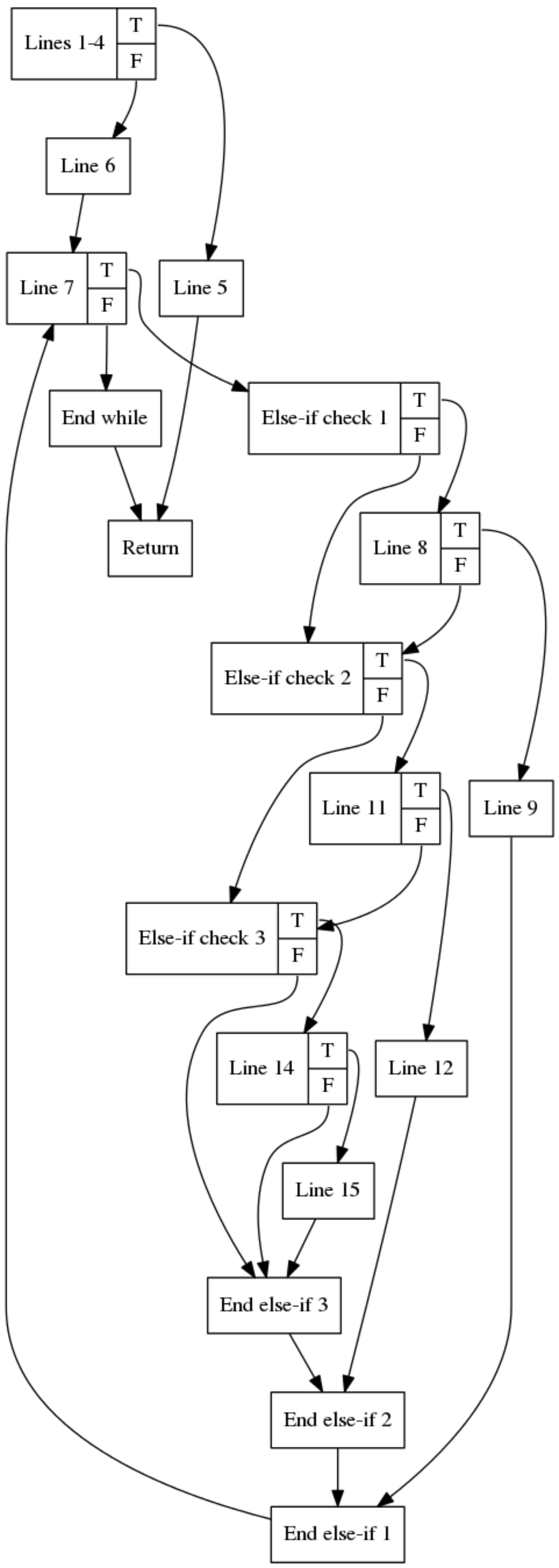

FIGURE 3.7: CFG for program *q*.

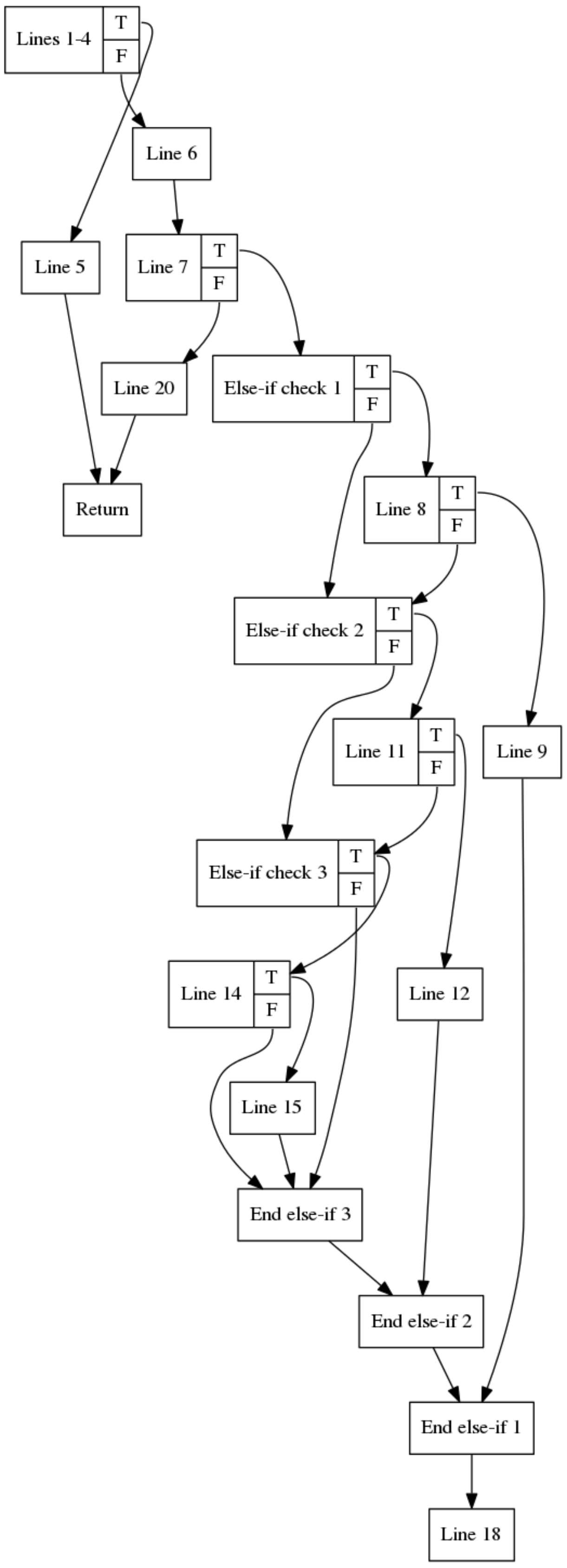

FIGURE 3.8: CFG for program $q_{transform}$

Chapter 4

Our Transformation Techniques

This chapter discusses the specifics of the optimizations we developed for this research, and how we arrived at them.

4.1 Basis for Transformations

We began by reasoning about common problems faced by symbolic execution and the structures that create these issues. We examined traditional symbolic execution, which uses time-limited depth-first search since complete symbolic execution is infeasible for programs of any significant size or complexity. Within this context, the goal is to maximize the amount of the program executed within the time limit.

Firstly, we considered the behavior of branching control flow paths. Depth-first search favors longer paths over shorter ones, regardless of their content or the number of sub-branches therein. Deep paths with many branches therefore take up more time during symbolic execution. This is not necessarily best for overall code coverage, as more lines of code may be contained in shorter paths.

Loops also cause significant trouble for symbolic execution [15], as symbolic depth-first search will execute the code within a loop as many times as is possible before examining any other part of the program. This can be a waste of time in terms of coverage, as no new statements are covered during that time if the loop iteration is not a factor in the control flow of the code contained therein.

4.2 Transformation Design and Implementation: The Slicer

Based off of these observations, we created several new program transformations, which are applied at compile time. Each has the general goal of syntactically reducing, which we term as *slicing*[1], a program to remove parts that have a low value proposition for symbolic execution. We introduce 5 slicing "modes" where each "mode" of our slicer is applied to the program under test on a function-by-function basis.

Most modes first explore a function's control-flow graph to find its longest acyclic source-sink path, and use the length of this *key path* as a guide while slicing the rest of the function. Identifying the key path can be a time-intensive process. It is a specific case of the *longest path problem,* which is an NP-Hard problem. It is at least fixed-parameter tractable in the length of the longest path d, meaning it can be solved in time $O(d!2^d n)$, where n is the number of nodes in the graph [31].

In our specific case, we are given a single source node from which to find paths in a directed graph, which significantly decreases the complexity. Our algorithms use depth-first search that breaks on cycles and are thus worst-case linear in the number of edges in the graph even without a priori knowledge of the length of the key path. However, as functions can have hundreds of edges, the time requirements can still increase beyond the point of feasibility. As such, we included the option to limit the amount of time the optimizer spends doing so in each function in the program under test. At the end of the specified time, the optimizer slices the function with regard to the longest path found in that time. Figures 4.1, 4.2, 4.3, 4.4, and 4.5 describe the five modes using pseudo-code.

Mode 1: The first of these transformations finds all acyclic source-sink paths in the function and slices each of them to half of their length. When path lengths are uneven (i.e. a path of length X where X mod 2 = 1) the optimizer leaves behind a path of length $ceiling(X/2)$. The same holds true throughout the slicer with regard to uneven path lengths. Due to state space explosion, the number of such paths can grow exponentially with the size of the program. Thus, this mode does not scale to real-world programs.

Mode 2: The second mode identifies the longest acyclic source-sink path and slices that path to half of its length. It then repeats this process until there is no acyclic source-sink path with

[1]Our use of the term slicing is different from the common use of this term in the context of *program slicing* where a slice is created with respect to a given set of variables [30].

```
identify all acyclic source-sink paths {
 recursively traverse graph using DFS
 upon finding a cycle or sink {
  if sink: store path, return
  if cycle: return
 }
 for each identified path {
   slice path in half {
    x = length of path/2
    l = length of path
    remove last l-x nodes from path
    add return block to end of path
 }
}
```

FIGURE 4.1: Pseudocode for Mode 1.

```
identify key path {
 while(time has not expired) {
  recursively traverse graph using DFS
  upon finding a cycle or sink {
   if sink:
    if current path longer current longest path:
     overwrite current longest path with
      current path, return
    else return
   if cycle: return
  }
 }
}
l = length of key path
x = length of key path/2
slice key path {
    remove last l-x nodes from key path
    add return block to end of key path
}
while(time has not expired) {
    identify key path
    if key path length <= x: break
    slice key path
}
```

FIGURE 4.2: Pseudocode for Mode 2.

length greater than half of the key path's original length or until time runs out. If the key path cannot be definitively identified in the allotted time, this mode is identical to the third. However, as discussed in Section 5.5, this mode can produce superior results even with an identical time budget.

Mode 3: The third transformation is a reduced version of the second. It similarly identifies the

```
identify key path
slice key path
```

FIGURE 4.3: Pseudocode for Mode 3.

```
identify key path
x = length of key path/2
blind slice {
 recursively traverse graph using DFS
 after making x traversals: direct all outgoing
  edges of current node to return blocks, return
}
```

FIGURE 4.4: Pseudocode for Mode 4.

```
loop slice {
 while(time has not expired) {
  recursively traverse graph using DFS
  upon finding a cycle: direct cyclic outgoing
   edge of current node to return block, return
 }
}
```

FIGURE 4.5: Pseudocode for Mode 5.

key path and slices it in half, but stops there. This is a minor optimization, but it nonetheless provides a slight advantage for symbolic execution and fares well in some cases.

Mode 4: The fourth optimization works similarly to the second, but instead of slicing only acyclic source-sink paths, it blindly slices all paths to half of the key path's length. In essence, this mode ensures that there are no paths of length greater than $ceiling(X/2)$, where X is the original length of the key path. This can be categorized as a more aggressive variant of traditional depth-limited symbolic execution. It removes nodes from the control flow graph entirely if there exists *any* path that could reach them in $ceiling(X/2)$ or more traversals, meaning that nodes that could be reached in depth-limited execution through shorter paths can be removed from consideration in programs sliced using this mode.

Mode 5: The final transformation causes all loops to return after a single iteration. This massively decreases the number of paths through the graph and helps ensure that time is not wasted repeatedly executing the same section of code.

The slicer also has a "Mode 0" which does not apply any transformations. This is a bookkeeping marker to identify the original program while keeping a consistent numbering scheme.

Chapter 5

Evaluation

In this study we seek to answer one core research question: "How do non-semantics-preserving program transformations impact the process of using symbolic execution for test generation and the quality of the tests thereby generated?"

5.1 Test Subjects

Unix's Coreutils are a standard group of programs used for research on symbolic execution [7, 22, 23], including the study on compiler optimizations that motivated this research. Specifically, we examined forty programs in Coreutils 6.11: *base64, basename, chcon, cksum, comm, cut, dd, dircolors, dirname, du, env, expand, expr, fold, groups, link, logname, mkdir, mkfifo, mknod, nice, nl, od, paste, pathchk, printf, readlink, rmdir, setuidgid, sleep, split, sum, sync, tee, touch, tr, tsort, unexpand, unlink,* and *wc.* We also created six small example programs on which we could reasonably run KLEE to completion: *double_loop,* two identical consecutive loops; *loop_switch,* a conditional structure inside a loop (both as described in Chapter 3); *double_cond,* two identical consecutive conditional statements; *get_sign,* a single 3-way conditional branch; *simple_switch,* a simple conditional structure; and *add_ints,* a simple function call.

5.2 Independent Variables

In our study, we adjusted the following independent variables:

Different transformations: We designed a number of optimizations for the study, as described in Section 4.2.

Caching: We aimed to simulate traditional symbolic execution, which is performed without caching. However, caching is a known method for reducing the calls to the SMT solver, which was one of the design goals for our optimizations. Thus, we also conducted the same experiments with caching enabled to observe the interactions.

While KLEE supports input bound definition (see Chapter 2), we chose not to provide any such limits to maximize the generality and reproducibility of our results.

5.3 Dependent Variables

As noted in Chapter 2, testability transformations can alter a program's test adequacy criteria. We chose not to do so in this study to maximize the generality of the optimizations. We examined the following dependent variables:

Change In Line Coverage: Through gcov, a coverage reporting tool compatible with LLVM [32], we were able to record the percentage of a program's code executed by a given test suite. By comparing the line coverage for different optimizations' test suites to that of the test suite for the non-optimized version of the program, we can see how optimizations affect that particular program's suitability for symbolic execution. From a broader perspective, by comparing the change in coverage from non-sliced to sliced for a given mode across multiple programs, we can get a sense of how well that optimization performs.

Execution Time: Faster execution time is naturally preferable during test generation. For the Coreutils, we limited the allowed execution time as is typical in traditional symbolic execution settings. If KLEE reported that symbolic execution finished before time ran out, it was noted. The time used by the slicer was also noted, to generate a complete picture of how much time test generation took for a given program. The "unsliced" original versions of the programs were given additional time for symbolic execution based on the maximum amount of time taken to generate their sliced counterparts to ensure fair comparisons.

SMT Solver Queries: As mentioned before, the key bottleneck in symbolic execution is the time spent by the SMT solver. For this reason, the number of queries issued is a key metric.

5.4 Methodology

Each of the proof-of-concept example programs was sliced using each of the slicer's modes. The resulting LLVM bitcode was fed to KLEE with a five-minute time limit. The original versions of the example programs were instrumented with gcov to provide a metric for test suite quality. We ran each of the generated test suites against its originating program and used gcov and KLEE's reports as our data source.

We used the same forty Coreutils used by Dong et. al. [23] in their study of traditional compiler optimizations for ease of comparison. The programs were again instrumented using gcov. Each of the forty programs was sliced using modes zero, two, three, four, and five using a 30-second per-function time limit, chosen to minimize the number of functions on which the slicer terminated prematurely. Mode one was excluded due to the aforementioned scaling issue. As before, we ran all of the resulting bitcode through KLEE with a five-minute time limit.

We repeated these experiments with caching enabled.

5.5 Results

The tables we provide in this chapter provide an illustrative subset of our total results. The full set of results can be found in Appendices A and B.

5.5.1 Small Examples

Symbolic execution of each of the small example programs described in Section 5.1 provides a test suite that gives 100% statement coverage within a small amount of time, with the exception of the deep conditional structure (93% coverage). Symbolic execution of the sliced versions of each program provided matching coverage in all but one case. Symbolic execution of sliced programs tends to be of shorter duration and involve fewer queries. In particular, the more complex programs (double_loop, loop_switch) show the greatest amount of speedup. This suggests that these optimizations are capable of increasing efficiency.

Enabling caching showed no change in coverage. This was expected, as the test suites already had 100% coverage and enabling caching was not likely to reduce KLEE's effectiveness. In the

Slicer Mode	Average Time (s)
2	97.7
3	92.6
4	183
5	98.7

TABLE 5.1: Average time taken by each slicer mode.

larger example programs, loop_switch and double_loop, it did facilitate a noticeable reduction both in the number of SMT solver queries and in execution time. The smaller examples saw a smaller and less consistent change in these values.

The full set of results for our small examples can be seen in Tables A.1 and A.2 in Appendix A.

5.5.2 Unix's Coreutils

As these programs are massively more complex than our examples, the time spent in the optimizer becomes nontrivial. If the transformation process took just as much time as symbolic execution itself, it would hardly be of value. Our starting time budget of thirty seconds per function proved to be adequate in the majority of cases. Across all combinations of modes and programs, the optimization process took no longer than 280 seconds. The average time across all modes was 118 seconds. Table 5.1 shows the average time taken for each transformation. Mode 4's increased time is due to the fact that the time budget only limits path identification time, not slicing time, for modes 2, 3, and 4. Mode 5 performs these two processes simultaneously, and as such its slicing time is limited. As mode 4 must slice many more paths than modes 2 or 3, it consistently took far longer to complete. Most functions were sliced well within the time budget; the majority of programs had two or three large functions that used all of their allotted time.

Table 5.2 shows some notable examples from our results: the greatest improvement, worst loss, and example average cases for each mode. The "Comparison To Original" column shows what percentage of the original program's test suite's coverage the sliced program's test suite achieved. We used this metric as opposed to a raw difference in percentage or number of lines covered to provide more readable results and describe the patterns generally across all programs.

Symbolic execution times were generally reduced by the application of optimizations. Table 5.3 shows how many programs finished before their allotted time ran out for each slicer

Program	Mode	Coverage (%)	Slicer Time	Comparison To Original (%)	Execution Time (s)
dd	2	43.32	123	418	74.82
printf	4	17.9	184	23.47	4.13
tee	2	82.61	90	109.62	300
pathchk	3	64.39	83	108.97	300
od	4	60.62	183	120.73	300
rmdir	5	68.06	90	119.53	26.14

TABLE 5.2: Notable coverage change results.

Slicer Mode	Programs Finished Early
Overall	78
0	4
2	13
3	12
4	18
5	31

TABLE 5.3: Number of programs with execution times noticeably decreased.

Slicer Mode	Average Change in Coverage %
Overall	1.70
0	0.73
2	2.62
3	2.59
4	2.13
5	0.55

TABLE 5.4: Average change in coverage percentage after enabling caching.

mode. For all but one case in which the unsliced program finished before time ran out, the sliced programs finished faster. The exception to this rule was the "split" program, in which the executions on sliced programs went to time, but produced much better test suites.

After enabling caching, we saw some small changes in line coverage, both positive and negative. In 121 of 160 cases across the entire experiment, enabling caching caused no change in line coverage. Table 5.4 provides a breakdown of the change in coverage by slicer mode. The number of queries was changed much more significantly, with all but two trials using many fewer queries. Table 5.5 shows the average change in number of queries by slicer mode.

Enabling caching also brought down execution times in all trials whose non-cached versions finished before their time ran out. Specifically, the cached versions of these trials finished in 42% of the time of their non-cached counterparts on average. Five additional trials were

Slicer Mode	Average Change in # Queries
Overall	-13845
0	-25687
2	-13980
3	-13819
4	-11973
5	-5534

TABLE 5.5: Average change in the number of queries after enabling caching.

Slicer Mode	Average % of Non-Cached Time
Overall	42.12
0	32.05
2	45.69
3	53.44
4	40.20
5	38.66

TABLE 5.6: Average change in execution time after enabling caching.

only able to finish within their time budget when caching was enabled. Table 5.6 shows the breakdown by slicer mode.

Full results for the tested Coreutils with and without caching are shown in Table B.1 and B.2 respectively.

5.6 Analysis

5.6.1 Overall Performance

Across the 160 experimental trials in the non-cached version of the experiment, the optimized programs produced equivalent or superior results in 96 cases. Across the 160 trials in the cached version, the optimized programs did even better, producing equal- or higher-quality test suites in 103 cases. Table 5.7 shows the breakdown of the number of programs improved by each optimization. The combination of these cases provided improved coverage on 36 of the 40 tested Coreutils. In all but 8 trials where coverage was improved in each version of the experiment, the total time taken by the optimizer and KLEE to produce a test suite for a program was less than or equal to the time used for standard symbolic execution of the original program.

Slicer Mode	Executions Improved (Cache Disabled)	Executions Improved (Cache Enabled)
Overall	96	103
2	22	24
3	18	21
4	23	25
5	33	33

TABLE 5.7: Number of Coreutils improved by each slicer mode

Slicer Mode	Avg Coverage Change	Avg Coverage Gain Case	Avg Coverage Loss Case
2	109.56	130.49	74.66
3	95.59	112.53	72.82
4	110.04	125.75	74.98
5	123.08	134.96	67.11

TABLE 5.8: Average changes in coverage by slicer mode for Coreutils, cache disabled.

5.6.2 Caching Disabled

Table 5.8 shows the average coverage as compared to unsliced programs for each mode overall, when coverage was improved, and when it was worse. Slicer mode 5, which causes loops to terminate after one execution, was by far the most effective of our optimizations, with many more improved cases and the highest average increase over the unaltered programs. Mode 3, which only slices a single path, was the least effective. This is understandable, as it is only a minor optimization, but it still outperformed the others in two trials, suggesting that there are cases where fewer alterations achieve superior results.

When coverage was worse, the amount by which it suffered varied significantly, from a minimum of 23.47% of the unsliced program's test suite's coverage to a maximum of 97.7%. On average, optimized programs that performed poorly had 73.12% of their progenitor's coverage.

The difference in the number of queries used by the sliced and unsliced versions of the program has a clear relationship to the change in coverage. Table 5.9 shows a comparison between the number of queries made during the execution of each sliced program to the number made executing the unsliced programs by mode. To reduce the impact of outliers, the single greatest and least values for each mode were removed from this calculation. A pattern is apparent in modes 2 and 4: both made more queries than the original program in successful trials and fewer in unsuccessful ones. Meanwhile, mode 3 consistently made fewer

Slicer Mode	Avg Query Change	Avg Query Change (Coverage Gain)	Avg Query Change (Coverage Loss)
Overall	83.29	91.11	68.43
2	92.57	106.09	73.97
3	82.86	92.50	68.89
4	100.99	117.83	71.68
5	57.19	59.84	43.04

TABLE 5.9: Average changes in queries by slicer mode for Coreutils, cache disabled.

queries than either of its cousins. Mode 5 breaks from this pattern: it made far fewer queries across all trials, issuing about half as many during successful trials and fewer during unsuccessful ones.

The optimizations all make fewer queries in unsuccessful trials on average; 51 of the 64 cases that produced inferior test suites used fewer queries than their unsliced originators (this ratio is consistent across all slicer modes within a 1-case tolerance).

5.6.3 Caching Enabled

As referenced in Section 5.5.2, changes in line coverage were not large or consistent, but they were significant enough to increase the number of trials in which the optimized programs outperformed the non-optimized ones. In three cases, programs whose coverage was improved by optimization in the non-cached version did not see improvement in the cached version. In all of these cases, the original program had a larger increase in coverage than the optimized version with the addition of caching. Broadly speaking, caching made the optimizations more effective; Table 5.10 shows the average changes in coverage by slicer mode in this setting.

Enabling caching brought about a reduction in solver queries and execution time, as seen in Table 5.11. The ratio of number of queries made while executing the sliced versions of the programs to the number made while executing the originals also saw a significant increase; Table 5.12 shows these changes, it should be noted that the averages again do not include the top and bottom values to reduce the influence of outliers.

Slicer Mode	Avg Coverage Change	Avg Coverage Gain Case	Avg Coverage Loss Case
2	112.59	136.60	74.66
3	98.87	117.48	72.82
4	112.56	131.23	74.98
5	122.55	134.85	67.11

TABLE 5.10: Average changes in coverage by slicer mode for Coreutils, cache enabled.

Slicer Mode	Average % of Non-Cached Queries	Average % of Non-Cached Execution Time
Overall	42.66	42.12
0	46.86	32.05
2	44.30	45.69
3	43.20	53.44
4	42.72	40.20
5	35.37	38.66

TABLE 5.11: Average changes in queries and execution time by slicer mode with caching for Coreutils.

Slicer Mode	Avg % of Unsliced Queries	Avg % of Unsliced Queries (Coverage Gain)	Avg % of Unsliced Queries (Coverage Loss)
Overall	349.53	400.14	248.43
2	430.89	557.02	236.75
3	358.35	446.45	239.83
4	335.64	326.71	372.22
5	270.35	314.36	33.78

TABLE 5.12: Average changes in queries by slicer mode for Coreutils, cache enabled.

5.7 Threats to Validity

Threats to internal validity: There are multiple axes along which the parameters of this experiment could be adjusted. For example, we only ran KLEE on our subject programs with a 5-minute time budget, with the budget for the original programs adjusted by the amount of time required for slicing. Allowing more time for symbolic execution may demonstrate a ceiling for the coverage provided by test suites generated by symbolically executing the sliced programs that is not necessarily present for unsliced programs. Additionally, the time allotment for the slicer was held constant across all programs and modes, where altering it may have been more effective. In particular, for large functions, if the longest acyclic source-sink path is not identified within the allotted time, slicer modes 2 and 3 can be equivalent. Further experimentation altering these values is certainly advisable.

Threats to external validity: Our results show that the optimizations developed herein are not applicable for every program. To attempt to make this study as reproducible and generalizable as possible, we used a symbolic execution engine, KLEE, and group of programs, Coreutils, whose interactions are well-studied and which provide a broad variety of different use cases for symbolic execution. By design, our research is only a first step in the exploration of the possibilities offered by non-semantics preserving optimizations, and while we believe it to be a positive one, further experimentation and exploration using different optimizations, programs, and symbolic execution engines is needed.

Threats to construct validity: The standout threat to construct validity is the possibility that the chosen metrics used to measure relative performance do not provide an accurate representation thereof. To mitigate this threat, we used several different metrics, relied on the well-studied and broadly-used tools KLEE and gcov, and limited symbolic execution times for our subjects.

Chapter 6

Related Work

Symbolic execution [1, 2] has been the focus of many research projects for over a decade. A number of these projects address the scalability issues of the method, however, they mostly use techniques that optimize it with respect to the original behaviors of the program under test. To our knowledge, we introduce the first technique that optimizes symbolic execution using program transformations that are unsound *by design*. Next, we briefly describe some key techniques from previous work on scaling symbolic execution.

Dong's study of the impact that semantics-preserving transformations (specifically standard compiler optimizations) have on symbolic execution [33] is the closest in spirit to our work and provides its inspiration. The study observed that, somewhat counter-intuitively, some compiler optimizations can actually slow down symbolic execution. Cadar's more recent new ideas paper [29] hypothesizes about the use of non-semantics-preserving transformations for symbolic execution. The central idea of using unsound program transforms to pre-process programs for more efficient test data generation was initially formalized by Harman et al [24].

Directed incremental symbolic execution [18] introduced the idea of using a static analysis for more efficient symbolic execution in the context of change, e.g., for regression checking. Yang et al. [21] memoise the run of symbolic execution on the program under test as a trie structure and re-use it for optimizing the next run of symbolic execution after a change to the program or the search depth parameter. Green [13] caches the constraints that are solved during symbolic execution in a database, which allows re-use of constraint solving results, in the spirit of KLEE's constraint caching [27].

Simple static partitioning [20] computes pre-conditions and uses them to distribute the exploration space of symbolic execution among different workers. ParSym [19] uses the nondeterminism in the exploration to create work units and distribute the overall workload. Ranged symbolic execution [22] captures the state of a run of symbolic execution using a concrete input. It uses two inputs to represent a sub-space, termed *range* of the exploration space, and distributes the workload by creating consecutive ranges. These ranges are explored by separate workers; work stealing is used for load balancing.

PREfix and PREfast [34] are among the first techniques to introduce the idea of compositional analysis for symbolic execution. A number of more recent techniques, e.g, SMART [35], SMASH [36], CompoSE [37], and others [38, 39] further developed the idea and showed its usefulness in scaling symbolic execution.

Godefroid and Luchaup [15] provide a way to deal with the path explosion problem in symbolic execution caused by loops by introducing summarizing the loop and inferring simple loop invariants. Our slicer also addresses this problem; however, our slicer's loop-affecting mode does not attempt to summarize or preserve the loop's behaviors, rather it directly changes the behavior of the loop by causing the function to return unconditionally after a single iteration.

Chapter 7

Conclusion

We believe that we found an exciting new avenue for research into the improvement of symbolic execution. We examined the use of non-semantics-preserving transformations to optimize programs for symbolic execution and thus produce higher-quality test suites more efficiently than in traditional settings. To do this, we designed four new optimizations using LLVM, which were then enabled in KLEE. Upon testing these optimizations on 40 of Unix's Coreutils, we found that we were able to achieve our goal in more than 50% of cases. Enabling caching also caused these optimizations to generally become more effective. Across all cases, we were able to improve performance in 90% of the tested programs. We believe that this is a promising approach for tackling the scalability issues of symbolic execution, and hope that our work serves as a basis for more effective applications of this well-known technique. Symbolic execution does not play by the same rules as concrete execution, and our results here show that the optimizations designed for it do not need to do so either.

Appendix A

Example Results

Program	Slicer Mode	Coverage (%)	Time (s)	Queries
double_loop	0	100	324.79	1474
	1	100	306.48	1106
	2	100	306.93	1137
	3	100	305.36	1135
	4	100	6.19	188
	5	100	6.25	186
double_cond	0	100	71.51	1701
	1	100	80.19	1790
	2	100	76.65	1766
	3	100	77.31	1779
	4	100	77.31	1723
	5	100	72.87	1698
loop_switch	0	100	19.63	1619
	1	100	1.83	123
	2	100	1.9	124
	3	100	1.88	125
	4	100	1.9	124
	5	100	3.11	196
simple_switch	0	93	3.58	208
	1	53	1.88	127
	2	93	2.2	148
	3	93	2.47	170
	4	93	2.26	148
	5	93	3.43	211
add_ints	0	100	6.07	188
	1	100	5.99	186
	2	100	6.01	186
	3	100	5.97	186
	4	100	5.86	184
	5	100	6.11	186
get_sign	0	100	2.33	160
	1	100	2.01	124
	2	100	2.05	127
	3	100	1.98	125
	4	100	1.84	123
	5	100	2.36	159

TABLE A.1: Coverage, Query, and Execution Time Results for small example programs, cache disabled

Program	Slicer Mode	Coverage (%)	Time (s)	Queries
double_loop	0	100	300	1205
	1	100	300	1046
	2	100	300	988
	3	100	300	987
	4	100	6.38	184
	5	100	6.39	185
double_cond	0	100	87.67	1765
	1	100	82.77	1811
	2	100	79.92	1728
	3	100	80.16	1741
	4	100	84.32	1803
	5	100	78.72	1673
loop_switch	0	100	18.63	1626
	1	100	1.87	125
	2	100	1.78	123
	3	100	1.8	124
	4	100	1.82	124
	5	100	2.92	193
simple_switch	0	93	3.67	215
	1	53	1.92	126
	2	93	2.29	151
	3	93	2.6	176
	4	93	2.35	150
	5	93	3.71	215
add_ints	0	100	6.28	186
	1	100	6.06	186
	2	100	6.00	184
	3	100	6.45	184
	4	100	6.27	184
	5	100	6.55	186
get_sign	0	100	2.21	162
	1	100	1.95	127
	2	100	1.97	127
	3	100	1.91	124
	4	100	1.94	123
	5	100	2.16	155

TABLE A.2: Coverage, Query, and Execution Time Results for small example programs, cache enabled

Appendix B

Appendix B Coreutils Results

Program	Slicer Mode	Slicer Time	Execution Time	Coverage (%)	% of Unsliced Coverage	Queries	% of Unsliced Queries
base64	0	0	473	70.48		78692	
	2	90	300	72.38	102.7	23243	29.54
	3	81	300	72.38	102.7	24017	30.52
	4	173	300	72.38	102.7	20999	26.69
	5	90	300	80	113.5	19481	24.76
basename	0	0	317	100		32470	
	2	90	5	92.31	92.31	686	2.11
	3	83	5	92.31	92.31	686	2.11
	4	178	5	92.31	92.31	686	2.11
	5	90	5.5	97.44	92.31	732	2.25
chcon	0	0	503	67.69		59624	
	2	120	300	51.28	75.75	38748	64.99
	3	113	300	51.28	75.75	38947	65.32
	4	203	300	51.28	75.75	39798	66.75
	5	120	65.76	43.59	64.4	8582	14.39
cksum	0	0	474	91.94		51671	
	2	90	32.74	85.48	92.97	3791	7.34

Program	Slicer Mode	Slicer Time	Execution Time	Coverage (%)	% of Unsliced Coverage	Queries	% of Unsliced Queries
	3	82	32.42	85.48	92.97	3791	7.34
	4	174	32.24	85.48	92.97	3791	7.34
	5	90	29.14	91.94	100	4256	8.24
comm	0	0	483	74.49		57977	
	2	90	300	53.06	71.23	47907	82.63
	3	84	300	51.02	68.49	47572	82.05
	4	183	21.24	67.35	90.14	3402	5.87
	5	90	27.7	81.63	109.6	3528	6.09
cut	0	0	480	50.34		31329	
	2	90	300	49.32	97.97	8683	27.72
	3	83	300	46.96	93.29	8796	28.08
	4	180	300	47.64	94.64	8793	28.07
	5	90	54.22	26.01	51.67	5752	18.36
dd	0	0	503	10.34		45408	
	2	123	74.82	43.32	419	7255	15.98
	3	113	300	4.63	44.78	51	0.11
	4	203	144.5	43.32	419	16359	36.02
	5	124	6.02	37.97	367.2	802	1.77
dircolors	0	0	472	69.47		15879	
	2	90	300	77.89	112.1	10943	68.91
	3	82	300	77.89	112.1	10784	67.91
	4	172	300	77.89	112.1	44185	278.3
	5	90	27.42	49.97	71.93	3837	24.16
dirname	0	0	186.16	100		21942	
	2	90	4.67	100	100	686	3.13
	3	81	4.56	100	100	686	3.13
	4	172	4.6	100	100	686	3.13

Program	Slicer Mode	Slicer Time	Execution Time	Coverage (%)	% of Unsliced Coverage	Queries	% of Unsliced Queries
	5	90	4.97	100	100	720	3.28
du	0	0	571	55.63		1686	
	2	194	300	73.18	131.5	52427	3109
	3	177	300	36.42	65.47	43918	2604
	4	271	300	73.51	132.5	48845	2897
	5	193	300	79.8	143.5	23831	1413
env	0	0	476	100		12051	
	2	90	300	88.89	88.89	32073	266.1
	3	81	300	88.89	88.89	31923	264.9
	4	176	300	97.78	97.78	54049	448.5
	5	90	191.24	100	100	13708	113.7
expand	0	0	472	39.07		49817	
	2	127	300	70.86	181.4	50345	101.1
	3	117	300	39.07	100	29731	59.68
	4	211	300	73.51	188.1	55585	111.6
	5	130	86.18	80.79	206.8	9804	19.68
expr	0	0	511	40.83		10473	
	2	127	4.38	32.54	79.69	654	6.24
	3	117	4.02	32.54	79.69	654	6.24
	4	211	4.41	32.54	79.69	654	6.24
	5	130	300	53.85	131.9	27900	266.4
fold	0	0	472	42.48		54834	
	2	90	300	58.41	137.5	27876	50.84
	3	82	300	58.41	137.5	27955	50.98
	4	172	300	52.21	122.9	15364	28.02
	5	90	300	79.65	187.5	21893	39.93
groups	0	0	471	94.59		27737	

Program	Slicer Mode	Slicer Time	Execution Time	Coverage (%)	% of Unsliced Cover-age	Queries	% of Unsliced Queries
	2	90	99.6	64.86	68.57	2347	8.46
	3	82	238.6	81.08	85.71	6262	22.58
	4	172	10.92	81.08	85.71	1421	5.12
	5	90	300	94.59	100	27401	98.79
link	0	0	474	71.43		80010	
	2	90	5.57	89.29	125	729	0.91
	3	82	5.42	85.71	119.9	692	0.86
	4	174	5.6	89.29	125	729	0.91
	5	90	6.35	96.43	135	803	1
logname	0	0	472	56		52568	
	2	90	4.81	92	164.3	799	1.52
	3	82	4.78	92	164.3	799	1.52
	4	172	4.56	92	164.3	799	1.52
	5	90	6.04	92	164.3	802	1.53
mkdir	0	0	472	60.61		40290	
	2	90	300	77.27	127.5	43396	107.7
	3	81	300	71.21	117.5	44335	110
	4	172	300	78.79	130	43757	108.6
	5	90	30	78.79	130	3792	9.41
mkfifo	0	0	476	72.34		57802	
	2	90	300	63.83	88.24	40303	69.72
	3	81	300	63.83	88.24	39211	67.84
	4	172	300	63.83	88.24	39339	68.06
	5	90	20	91.49	126.47	2701	4.67
mknod	0	0	490	47.56		53291	
	2	91	300	50	105.1	28658	53.78
	3	86	300	41.46	87.17	28579	53.63

Program	Slicer Mode	Slicer Time	Execution Time	Coverage (%)	% of Unsliced Coverage	Queries	% of Unsliced Queries
	4	190	300	50	105.1	28423	53.34
	5	92	21	65.85	138.5	2828	5.31
nice	0	0	480	76.27		34328	
	2	90	300	76.27	100	17343	50.52
	3	82	300	61.02	80	16458	47.94
	4	180	300	76.27	100	17434	50.79
	5	90	16	94.42	124.5	2133	6.21
nl	0	0	483	54.98		69860	
	2	93	300	49.76	90.51	27826	39.83
	3	86	300	51.18	93.09	25028	25.82
	4	183	300	46.45	84.49	16068	23
	5	96	300	75.83	137.9	16248	25.26
od	0	0	479	50.21		29619	
	2	90	300	51.48	102.5	38798	131
	3	82	300	59.49	118.5	19011	64.19
	4	179	300	60.62	120.7	18097	61.1
	5	90	0.32	27.99	55.75	79	0.27
paste	0	0	483	64.71		117721	
	2	90	300	58.82	90.9	30572	25.97
	3	83	300	58.82	90.9	29437	25.01
	4	183	14.89	59.89	92.55	37616	31.95
	5	90	40.4	71.66	110.7	2819	2.39
pathchk	0	0	472	59.09		38270	
	2	90	300	43.18	73.07	58190	152.1
	3	82	300	64.39	109	25650	67.02
	4	172	300	43.18	73.07	1983	5.18
	5	90	27.42	52.27	88.46	5537	14.47

Program	Slicer Mode	Slicer Time	Execution Time	Coverage (%)	% of Unsliced Cover-age	Queries	% of Unsliced Queries
printf	0	0	484	76.26		26144	
	2	90	4.77	17.9	23.47	649	2.48
	3	83	2.95	17.9	23.47	649	2.48
	4	184	4.13	17.9	23.47	649	2.48
	5	90	300	80.16	105.1	15043	57.54
readlink	0	0	494	100		65611	
	2	90	300	96	96	48784	74.35
	3	83	300	96	96	50029	76.25
	4	193	300	96	96	48918	74.56
	5	90	35.27	100	100	4657	7.1
rmdir	0	0	493	56.94		67222	
	2	90	300	56.94	100	55602	82.71
	3	83	300	75	131.2	45107	67.1
	4	178	24.69	66.67	117.1	3135	4.66
	5	90	26.14	68.06	119.5	3361	5
setuidgid	0	0	481	23.38		86465	
	2	90	300	23.38	100	11347	13.12
	3	83	300	23.38	100	11987	13.86
	4	181	300	23.38	100	12642	14.62
	5	90	300	23.38	100	13968	16.15
sleep	0	0	487	45.65		53013	
	2	90	4.61	63.04	138.09	808	1.52
	3	83	4.77	63.04	138.09	808	1.52
	4	187	4.73	63.04	138.09	808	1.52
	5	90	6.02	63.04	138.09	802	1.51
split	0	0	18	34.1		4092	
	2	90	300	47.47	139.2	32402	791.8

Program	Slicer Mode	Slicer Time	Execution Time	Coverage (%)	% of Unsliced Coverage	Queries	% of Unsliced Queries
	3	83	300	47.47	139.2	33534	819.5
	4	182	300	47.47	121.6	54376	1329
	5	90	300	58.53	171.6	36387	889.2
sum	0	0	550	86.23		33780	
	2	120	300	81.05	93.99	52630	155.8
	3	115	300	81.05	93.99	50648	152.9
	4	220	15.17	86.32	100.1	2136	6.32
	5	120	36.89	90.53	105	5469	16.19
sync	0	0	5.99	100		802	
	2	90	4.91	100	100	799	99.63
	3	85	1.85	100	100	799	99.63
	4	179	1.84	100	100	799	99.63
	5	90	5.92	100	100	802	100
tee	0	0	480	75.36		47201	
	2	90	300	82.61	109.6	54965	116.4
	3	82	300	76.81	101.9	46550	98.62
	4	180	20.47	84.06	111.5	2669	5.65
	5	90	20.69	86.96	115.4	2729	5.78
touch	0	0	495	59.72		11373	
	2	114	300	55.56	93.03	11505	101.2
	3	105	300	48.61	81.4	3900	34.29
	4	195	300	55.56	93.03	14506	127.5
	5	121	74.62	72.22	121	8924	78.47
tr	0	0	491	42.34		43422	
	2	90	300	18.36	43.36	49541	114.1
	3	81	300	18.36	43.36	44413	102.3
	4	172	300	16.54	39.06	48191	111

Program	Slicer Mode	Slicer Time	Execution Time	Coverage (%)	% of Unsliced Cover-age	Queries	% of Unsliced Queries
	5	90	79.71	45.37	107.2	10393	23.93
tsort	0	0	480	72.41		865	
	2	90	4.3	25.12	34.69	799	92.36
	3	83	14.55	27.09	37.42	1642	189.8
	4	180	4.95	25.12	34.69	775	89.59
	5	90	11.34	29.06	40.13	1597	184.6
unexpand	0	0	475	44.85		24101	
	2	90	300	60.31	134.5	62662	260
	3	83	300	40.21	89.65	20049	83.19
	4	175	300	47.94	106.9	64871	269.2
	5	90	102.5	84.02	187.3	10514	43.62
unlink	0	0	472	72		51014	
	2	90	5.61	100	138.9	799	1.57
	3	83	6.5	96	133.3	799	1.57
	4	172	6.59	96	133.3	799	1.57
	5	90	7.09	100	138.9	913	1.79
wc	0	0	473	55.34		25583	
	2	90	300	65.65	118.6	62310	243.6
	3	82	300	59.92	108.3	60624	237
	4	173	300	59.92	108.3	58929	230.3
	5	90	57.58	64.89	117.3	8199	32.05

TABLE B.1: Coverage, Query and Execution Time Results for Coreutils, Caching Disabled

Program	Slicer Mode	Slicer Time	Execution Time	Coverage (%)	% of Unsliced Coverage	Queries	% of Unsliced Queries
base64	0	0	473	70.48		7819	
	2	90	300	72.38	102.7	8602	110
	3	81	300	72.38	102.7	8927	114.2
	4	173	300	80	113.5	10974	1740.4
	5	90	170	80.95	114.9	7690	98.35
basename	0	0	48.6	100		1330	
	2	90	1.98	92.31	92.31	309	23.23
	3	83	1.98	92.31	92.31	309	23.23
	4	178	1.98	92.31	92.31	309	23.23
	5	90	2.14	97.44	92.31	336	25.26
chcon	0	0	503	67.69		9718	
	2	120	300	62.56	92.42	17221	177.2
	3	113	300	52.31	77.29	17220	177.2
	4	203	300	57.44	84.85	18988	195.4
	5	120	13.37	43.59	64.4	1637	16.85
cksum	0	0	272.5	91.94		51671	
	2	90	8.2	85.48	92.97	3791	7.34
	3	82	8.24	85.48	92.97	3791	7.34
	4	174	8.34	85.48	92.97	3791	7.34
	5	90	18.91	91.94	100	4256	8.24
comm	0	0	483	50.34		17184	
	2	90	300	67.35	90.41	22179	82.63
	3	84	300	46.96	68.49	21022	82.05
	4	183	6.49	81.63	109.6	1331	5.87
	5	90	10.3	81.63	109.6	1480	6.09
cut	0	0	480	56.76		6121	
	2	90	300	49.32	86.89	6242	102

Program	Slicer Mode	Slicer Time	Execution Time	Coverage (%)	% of Unsliced Coverage	Queries	% of Unsliced Queries
	3	83	300	46.96	82.73	6454	105.4
	4	180	300	47.64	83.93	6261	102.3
	5	90	8.16	26.01	45.82	860	14.05
dd	0	0	503	10.34		10745	
	2	123	40.09	43.32	419	2701	25.13
	3	113	300	4.63	44.78	44	0.41
	4	203	78.53	43.32	419	8407	78.24
	5	124	2.35	37.97	367.2	374	3.48
dircolors	0	0	472	69.47		7541	
	2	90	300	76.84	110.6	1649	21.87
	3	82	300	77.89	112.1	1800	23.87
	4	172	300	77.89	112.1	13539	179.53
	5	90	4.99	49.97	71.93	561	7.44
dirname	0	0	51.09	100		2104	
	2	90	2.03	100	100	309	14.69
	3	81	2.17	100	100	309	14.69
	4	172	2.41	100	100	309	14.69
	5	90	2.78	100	100	340	16.16
du	0	0	571	51.66		1071	
	2	194	300	81.79	158.32	29355	2740
	3	177	300	46.69	90.38	6251	583.7
	4	271	300	81.79	158.3	30195	2819
	5	193	75.27	79.47	153.8	4452	415.7
env	0	0	476	100		853	
	2	90	300	97.78	97.78	3043	356.7
	3	81	300	97.78	97.78	3472	407
	4	176	300	97.78	97.78	21971	2576

Program	Slicer Mode	Slicer Time	Execution Time	Coverage (%)	% of Unsliced Coverage	Queries	% of Unsliced Queries
	5	90	70.39	100	100	1148	134.6
expand	0	0	472	39.07		15797	
	2	127	300	70.86	181.4	40017	253.3
	3	117	300	39.07	100	10274	65.04
	4	211	300	73.51	188.1	37095	234.8
	5	130	34.55	80.79	206.8	3589	22.72
expr	0	0	511	40.83		9847	
	2	127	1.72	32.54	79.69	300	3.05
	3	117	1.63	32.54	79.69	300	3.05
	4	211	1.69	32.54	79.69	300	3.05
	5	130	300	57.99	142	2298	23.34
fold	0	0	472	42.48		196	
	2	90	300	56.64	133.3	17266	8809
	3	82	300	56.64	133.3	15659	7989
	4	172	248.5	63.72	150	6475	3304
	5	90	300	79.65	187.5	17070	8709
groups	0	0	408	94.59		1137	
	2	90	84.05	64.86	68.57	295	25.95
	3	82	176.1	81.08	85.71	375	32.98
	4	172	3.13	81.08	85.71	299	26.3
	5	90	196.34	94.59	100	1005	88.39
link	0	0	474	67.89		15309	
	2	90	2.65	89.29	131.6	305	1.99
	3	82	2.7	85.71	126.3	268	1.75
	4	174	2.7	89.29	131.6	305	1.99
	5	90	3.28	96.43	142.1	375	2.45
logname	0	0	472	56		10490	

Program	Slicer Mode	Slicer Time	Execution Time	Coverage (%)	% of Unsliced Cover-age	Queries	% of Unsliced Queries
	2	90	2.01	92	164.3	375	3.57
	3	82	2	92	164.3	375	3.57
	4	172	2.05	92	164.3	375	3.57
	5	90	3.12	92	164.3	374	3.57
mkdir	0	0	472	68.18		15032	
	2	90	300	71.21	104.44	12423	82.64
	3	81	300	71.21	104.44	12906	85.86
	4	172	300	78.79	115.6	13330	88.68
	5	90	6.63	78.79	115.6	720	4.79
mkfifo	0	0	476	72.34		10601	
	2	90	300	91.49	126.5	525	4.95
	3	81	300	85.11	117.7	5474	51.63
	4	172	300	85.11	117.7	5618	53
	5	90	20	91.49	126.47	525	4.95
mknod	0	0	490	47.56		10781	
	2	91	300	64.63	135.9	6842	63.46
	3	86	300	56.1	118	6720	62.33
	4	190	300	50	105.1	7409	68.72
	5	92	5.22	65.85	138.5	592	5.49
nice	0	0	480	76.27		13173	
	2	90	300	76.27	100	10578	80.3
	3	82	300	61.02	80	7839	59.51
	4	180	300	76.27	100	6595	50.06
	5	90	8.33	94.42	124.5	821	6.23
nl	0	0	483	54.98		1903	
	2	93	300	70.62	128.5	8021	421.5
	3	86	300	62.56	113.8	5096	267.8

Program	Slicer Mode	Slicer Time	Execution Time	Coverage (%)	% of Unsliced Coverage	Queries	% of Unsliced Queries
	4	183	300	60.66	110.3	10100	530.7
	5	96	300	77.25	140.5	10919	573.8
od	0	0	479	50.21		14402	
	2	90	300	51.48	102.5	22206	154.2
	3	82	300	67.37	134.2	17024	118.2
	4	179	300	65.4	130.3	23248	161.4
	5	90	0.14	27.99	55.75	37	0.26
paste	0	0	483	64.71		416	
	2	90	300	58.82	90.9	9145	2198
	3	83	300	58.82	90.9	9333	2244
	4	183	14.89	59.89	92.55	6927	1665
	5	90	4.77	71.66	110.7	527	126.7
pathchk	0	0	472	68.18		14788	
	2	90	300	43.18	63.33	36419	246.3
	3	82	300	65.91	96.67	7887	53.33
	4	172	3.1	43.18	63.33	357	2.41
	5	90	21.74	52.27	76.66	2517	17.02
printf	0	0	484	77.82		20202	
	2	90	1.63	17.9	23	295	1.46
	3	83	1.59	17.9	23	295	1.46
	4	184	1.6	17.9	23	295	1.46
	5	90	300	95.72	105.1	9336	46.21
readlink	0	0	494	100		25021	
	2	90	300	96	96	22600	90.22
	3	83	300	96	96	22425	89.52
	4	193	300	96	96	22712	90.66
	5	90	7.75	100	100	823	3.29

Program	Slicer Mode	Slicer Time	Execution Time	Coverage (%)	% of Unsliced Coverage	Queries	% of Unsliced Queries
rmdir	0	0	493	56.94		13329	
	2	90	300	63.89	112.2	32519	244
	3	83	300	75	131.2	15478	116
	4	178	4.95	66.67	117.1	531	3.98
	5	90	6.22	68.06	119.5	593	4.45
setuidgid	0	0	481	23.38		13864	
	2	90	300	23.38	100	6494	46.84
	3	83	300	23.38	100	6514	46.98
	4	181	300	23.38	100	7010	50.56
	5	90	300	23.38	100	10615	76.57
sleep	0	0	487	45.65		11317	
	2	90	2.02	63.04	138.09	384	3.39
	3	83	2.09	63.04	138.09	384	3.39
	4	187	2.06	63.04	138.09	384	3.39
	5	90	3.12	63.04	138.09	374	3.30
split	0	0	6.06	34.1		177	
	2	90	300	47.47	139.2	14143	7990
	3	83	300	47.47	139.2	14222	8035
	4	182	300	47.47	121.6	19795	11180
	5	90	300	58.53	171.6	14553	8222
sum	0	0	431	86.32		30069	
	2	120	300	78.95	91.46	24080	80.08
	3	115	300	78.95	91.46	22634	75.27
	4	220	3.47	86.32	100	458	1.52
	5	120	27.16	90.53	104.9	2156	7.17
sync	0	0	3.1	100		374	
	2	90	2.04	100	100	375	100.3

Program	Slicer Mode	Slicer Time	Execution Time	Coverage (%)	% of Unsliced Coverage	Queries	% of Unsliced Queries
	3	85	2.13	100	100	375	100.3
	4	179	2	100	100	375	100.3
	5	90	2.92	100	100	374	100
tee	0	0	448	75.36		25506	
	2	90	300	82.61	109.6	32560	127.7
	3	82	300	76.81	101.9	17792	69.76
	4	180	4.08	84.06	111.5	411	1.61
	5	90	4.32	86.96	115.4	471	1.85
touch	0	0	495	59.72		10581	
	2	114	300	55.56	93.03	5600	52.93
	3	105	300	48.61	81.4	3124	29.52
	4	195	300	55.56	93.03	9770	92.34
	5	121	26.61	72.22	121	2681	25.34
tr	0	0	491	42.34		211445	
	2	90	300	18.36	43.36	17250	8.16
	3	81	300	18.36	43.36	15170	7.17
	4	172	300	16.54	39.06	16186	7.65
	5	90	19.79	45.68	107.8	1069	0.51
tsort	0	0	480	72.41		729	
	2	90	2.07	25.12	34.69	375	51.44
	3	83	9.56	27.09	37.42	800	109.7
	4	180	2.08	25.12	34.69	351	48.15
	5	90	7.3	29.06	40.13	890	122.1
unexpand	0	0	475	56.7		23284	
	2	90	300	60.31	106.4	39460	169.5
	3	83	300	58.76	103.6	20229	86.88
	4	175	300	47.94	84.55	68929	167.2

Program	Slicer Mode	Slicer Time	Execution Time	Coverage (%)	% of Unsliced Cover-age	Queries	% of Unsliced Queries
	5	90	34.18	84.02	148.2	3498	15.02
unlink	0	0	472	72		11090	
	2	90	2.89	100	138.9	375	3.38
	3	83	2.87	96	133.3	375	3.68
	4	172	2.93	96	133.3	375	3.38
	5	90	3.64	100	138.9	431	3.89
wc	0	0	473	55.34		16549	
	2	90	300	68.32	123.5	26266	158.7
	3	82	300	62.6	113.1	17865	108
	4	173	300	59.92	108.3	18203	110
	5	90	19.61	64.89	117.3	2288	13.83

TABLE B.2: Coverage, Query and Execution Time Results for Coreutils, Caching Enabled

Bibliography

[1] L. A. Clarke, "A system to generate test data and symbolically execute programs," *TSE*, no. 3, pp. 215–222, 1976.

[2] J. C. King, "Symbolic execution and program testing," *Communications ACM*, vol. 19, no. 7, 1976.

[3] M. Papadakis and N. Malevris, "Automatic mutation test case generation via dynamic symbolic execution," in *2010 IEEE 21st International Symposium on Software Reliability Engineering*, pp. 121–130, Nov 2010.

[4] T. Su, Z. Fu, G. Pu, J. He, and Z. Su, "Combining symbolic execution and model checking for data flow testing," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 654–665, May 2015.

[5] P. Muntean, V. Kommanapalli, A. Ibing, and C. Eckert, *Computer Safety, Reliability, and Security: 34th International Conference, SAFECOMP 2015, Delft, The Netherlands, September 23-25, 2015, Proceedings*, ch. Automated Generation of Buffer Overflow Quick Fixes Using Symbolic Execution and SMT, pp. 441–456. Cham: Springer International Publishing, 2015.

[6] L. de Moura and N. Bjørner, *Formal Methods: Foundations and Applications: 12th Brazilian Symposium on Formal Methods, SBMF 2009 Gramado, Brazil, August 19-21, 2009 Revised Selected Papers*, ch. Satisfiability Modulo Theories: An Appetizer, pp. 23–36. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

[7] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pp. 209–224, 2008.

[8] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley., "Automatic exploit generation," in *NDSS*, 2011.

[9] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," *SIGPLAN Not.*, vol. 40, pp. 213–223, June 2005.

[10] S. Khurshid, C. S. Păsăreanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'03, (Berlin, Heidelberg), pp. 553–568, Springer-Verlag, 2003.

[11] D. Molnar, X. C. Li, and D. Wagner, "Dynamic test generation to find integer bugs in x86 binary linux programs," in *USENIX Security*, pp. 67–82, 2009.

[12] S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke, "Combining symbolic execution with model checking to verify parallel numerical programs," in *TOSEM*, vol. 17, pp. 1–10, 2008.

[13] W. Visser, J. Geldenhuys, and M. B. Dwyer, "Green: Reducing, reusing and recycling constraints in program analysis," in *FSE*, 2012.

[14] C. Cadar, P. Godefroid, S. Khurshid, C. S. Pasareanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice: Preliminary assessment," in *ICSE*, pp. 1066–1071, Springer, 2011.

[15] P. Godefroid and D. Luchaup, "Automatic partial loop summarization in dynamic test generation," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, (New York, NY, USA), pp. 23–33, ACM, 2011.

[16] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *ASE*, pp. 443–446, 2008.

[17] Y. Li, Z. Su, L. Wang, and X. Li, "Steering symbolic execution to less traveled paths," in *OOPSLA*, pp. 19–32, 2013.

[18] S. Person, G. Yang, N. Rungta, and S. Khurshid, "Directed Incremental Symbolic Execution," in *PLDI*, pp. 504–515, 2011.

[19] J. H. Siddiqui and S. Khurshid, "ParSym: Parallel symbolic execution," in *ICSTE*, pp. V1–405–V1–409, Oct. 2010.

[20] M. Staats and C. Păsăreanu, “Parallel Symbolic Execution for Structural Test Generation,” in *ISSTA*, pp. 183–194, 2010.

[21] G. Yang, S. Khurshid, and C. S. Păsăreanu, “Memoise: A tool for memoized symbolic execution,” in *Proceedings of the 2013 International Conference on Software Engineering*, ICSE ’13, (Piscataway, NJ, USA), pp. 1343–1346, IEEE Press, 2013.

[22] J. H. Siddiqui and S. Khurshid, “Scaling symbolic execution using ranged analysis,” *SIGPLAN Not.*, vol. 47, pp. 523–536, Oct. 2012.

[23] S. Dong, O. Olivo, L. Zhang, and S. Khurshid, “Studying the influence of standard compiler optimizations on symbolic execution,” in *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*, pp. 205–215, Nov 2015.

[24] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper, “Testability transformation,” *IEEE Transactions on Software Engineering*, vol. 30, pp. 3–16, Jan 2004.

[25] “The llvm compilation infrastructure.” `http://llvm.org/`.

[26] C. Lattner and V. Adve, “Llvm: A compilation framework for lifelong program analysis & transformation,” in *CGO*, pp. 75–86, 2004.

[27] “The klee symbolic virtual machine.” `http://klee.github.io/klee`.

[28] “Klee coreutils study.” `http://klee.github.io/klee/TestingCoreutils.htmls`.

[29] C. Cadar, “Targeted program transformations for symbolic execution,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pp. 906–909, 2015.

[30] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen, “A brief survey of program slicing,” *SIGSOFT Softw. Eng. Notes*, vol. 30, pp. 1–36, Mar. 2005.

[31] H. Bodlaender, “On linear time minor tests with depth-first search,” *Journal of Algorithms*, vol. 14, no. 1, pp. 1 – 23, 1993.

[32] “The gnu coverage tool.” `http://gcc.gnu.org/onlinedocs/gcc/Gcov.html`.

[33] S. Dong, “Studying the influence of standard compiler optimizations on symbolic execution,” Master’s thesis, University of Texas at Austin, Austin, Texas, 5 2015.

[34] W. R. Bush, J. D. Pincus, and D. J. Sielaff, "A static analyzer for finding dynamic programming errors," *Software: Practice and Experience*, vol. 30, no. 7, pp. 775–802, 2000.

[35] P. Godefroid, "Compositional dynamic test generation," in *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, (New York, NY, USA), pp. 47–54, ACM, 2007.

[36] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. D. Tetali, "Compositional may-must program analysis: Unleashing the power of alternation," *SIGPLAN Not.*, vol. 45, pp. 43–56, Jan. 2010.

[37] R. Qiu, G. Yang, C. S. Pasareanu, and S. Khurshid, "Compositional symbolic execution with memoized replay," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 632–642, May 2015.

[38] S. Anand, P. Godefroid, and N. Tillmann, *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, ch. Demand-Driven Compositional Symbolic Execution, pp. 367–381. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[39] E. Albert, M. Gómez-Zamalloa, J. M. Rojas, and G. Puebla, *Logic-Based Program Synthesis and Transformation: 20th International Symposium, LOPSTR 2010, Hagenberg, Austria, July 23-25, 2010, Revised Selected Papers*, ch. Compositional CLP-Based Test Data Generation for Imperative Languages, pp. 99–116. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.