10th IEEE International Conference on Software Testing, Verification and Validation

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

Hayes Converse
Electircal and Computer Engineering
University of Texas at Austin
Austin, Texas
Email: hayesconverse@utexas.edu

Oswaldo Olivo
Computer Science
University of Texas at Austin
Austin, Texas
Email: olivo@cs.utexas.edu

Sarfraz Khurshid
Electrical and Computer Engineering
University of Texas at Austin
Austin, Texas
Email: khurshid@ece.utexas.edu

***Abstract*—Symbolic execution is a well-studied method that has a number of useful applications, including generation of high-quality test suites that find many bugs. However, scaling it to real-world applications is a significant challenge, as it depends on the often expensive process of solving constraints on program inputs. Our insight is that when the goal of symbolic execution is test generation, *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 five such transformations based on a few different program simplification heuristics that are designed to lower the cost of symbolic execution for input generation. As enabling technology we use the KLEE symbolic execution engine and the LLVM compiler infrastructure. We evaluate our transformations using a suite of small subjects as well as a subset of the well-studied Unix Coreutils. In a majority of cases, our approach reduces the time for symbolic execution for input generation and increases code coverage of the resultant suite.**

***Index Terms*—Symbolic Execution, compiler optimizations, non-semantics-preserving transformations, test generation, LLVM, KLEE.**

I. Introduction

Symbolic execution [1], [2], [3], [4] 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 [5], [6], [7], [8]. 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. For each path explored, a *path condition* is built to capture the logical constraints on inputs that execute that path. One common application of symbolic execution is test generation where the path conditions are solved using off-the-shelf solvers, e.g., SMT solvers [9], and the solutions are reified into concrete tests to create large, high-quality test suites [1], [10], [11], [4], [3], [2], [12], [13], [14]. 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 infeasibility. This makes symbolic execution impractical for many real-world applications [8].

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, Olivo, Zhang, and Khurshid [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. This approach was initially proposed by Cadar [24], but to date no concrete implementations and evaluations have been produced.

In an attempt to make symbolic execution faster while preserving its advantages, we present a new series of non-semantics-preserving *testability* transformations [25] 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 as our symbolic execution engine in this paper, as its foundation on LLVM allows us to easily create and use new compile-time program transformations. We submit our study of several new transformations designed end-to-end to improve symbolic execution in its traditional context (i.e., using time-limited depth-first search without caching). The design of these transformations was inspired by two guiding heuristics. Firstly, reducing the number of paths through a control-flow graph is likely to provide faster symbolic execution, as the state space has been reduced. Secondly, shorter paths likely have less-complex path conditions. A "complex" path condition is one which takes a large or infeasible amount of time in the SMT solver; certain operations (like integer multiplication) drastically increase the complexity of path conditions. We make a probabilistic argument in support of this heuristic. As a precondition, we establish that any extension to an existing path will have a path condition of equal or greater complexity than that of the original. Extensions to paths are made through branching; we argue that a path with more branches (i.e., one that is longer) is more likely to have a high-complexity operation than one with fewer. Thus, our transformations seek to decrease the number and length of paths through a program's control-flow graph.We evaluated these optimizations on a subset of Unix's Coreutils, a well-studied group of programs, 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 paper 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.
- Proof that such transformations can improve the execution time and line coverage of the programs under test.

Our work explores only a small space of the possible non-semantics-preserving transformations that can help scale symbolic execution. Our results may not compare favorably to more mature techniques, such as PEX [26], -Overify [27] or SAGE [28], but such comparisons are not presently our goal. This paper merely seeks to establish that this technique can, with only the most intuitive of transformations, provide faster symbolic execution for equal or higher-quality test suites. 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.

II. Background

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.

A. Symbolic Execution

Symbolic execution treats inputs to a program 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 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 as the analysis proceeds down a path. Thus, much of the time consumed by symbolic execution is spent in the SMT solver, heavily weighted toward the end of paths as SMT solving commonly becomes exponentially harder as the number of clauses in a Boolean formula increases.

In addition, the number of paths grows exponentially with the complexity of a program in what is called the path explosion problem. This presents a key challenge as each of the formulas associated with a path must be solved to ensure that the path is feasible. If it is, a solution to that formula becomes a test input. As noted above, there have been a number of approaches developed with the goal of addressing this bottleneck, both in the fields of SAT solving and symbolic execution. However, all present approaches do their best not to change the *semantics*, or input-output behavior, of the program under test.

B. LLVM and KLEE

The LLVM framework [29], [30] 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 [31], 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 [10], [32], 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.

C. Compiler Optimizations and Testability Transformations

Most compiler optimizations are *semantics-preserving 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. An 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 an 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. These transformations fall under the heading of *testability transformations,* [25], [33] 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.*

We here use the term "semantics" is here used to refer specifically to the traditional notion of semantics in programming, i.e. a program's input-output relationships. Harman notes that there is a *test semantics* that *is* preserved in the construction of these transformations [34]. Defining this semantics is an open problem in the field of testability transformations, and one whose solution may provide important fundamental understandings key to the design of new transformations.

KLEE's developers theorized about the possibility of using non-semantics-preserving transformations to improve symbolic execution [24], but this paper is the first to our knowledge to implement these testability transformations explicitly for this purpose.

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 inputs to the program. A mismatch between the resulting program and its parent 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 reduced program must have the same return type as its source, although the return value can vary arbitrarily. Finally, the program must still have an entrance point, so the first basic block must not be removed.

III. Motivating Examples

This section provides several examples using small programs to demonstrate how symbolic execution can be optimized using non-semantics-preserving transformations.

The program in Fig. 1 has two consecutive loops with identical, finite loop conditions. The second loop is executed with no regard to the behavior of the first. Thus, removing the second loop, as in Fig. 2, makes symbolic execution faster without decreasing the test suite's coverage. Control-flow graphs for both programs can be seen in Figs. 3 and 4, respectively. Making this alteration by hand reduces KLEE's execution time from 10.9 hours to 23 minutes, and reduces the number of queries issued to the SMT solver by 8672, from 15495 to 6823. Moreover, the maximum complexity of solver queries is also greatly reduced: the maximum depth of the pre-slice graph is $10n + 3$, where n is the maximum number of iterations of each loop, whereas the post-slice graph only has a maximum path length (and thus maximum solver query length) of $5n+3$. In this example, restricting this depth removes the most expensive part of symbolic execution. This program alteration therefore achieves the goal of depth-limited symbolic execution in a more flexible fashion. This result is achievable using one of our transformations, described in Section IV-B.

The program in Fig. 5 has a conditional structure inside of a loop. However, multiple 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 Fig. 6, we can significantly reduce the complexity of the structure while still providing 100% line coverage. The control-flow graphs for these programs (in Figs. 7 and 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, and makes only 201 solver queries for the reduced program as opposed to 1619 for the original. This is similarly achievable using one of our transformations.

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 V-D.

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

Fig. 1. A program with two repetitious loops.

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

Fig. 2. The same program with the second loop removed.

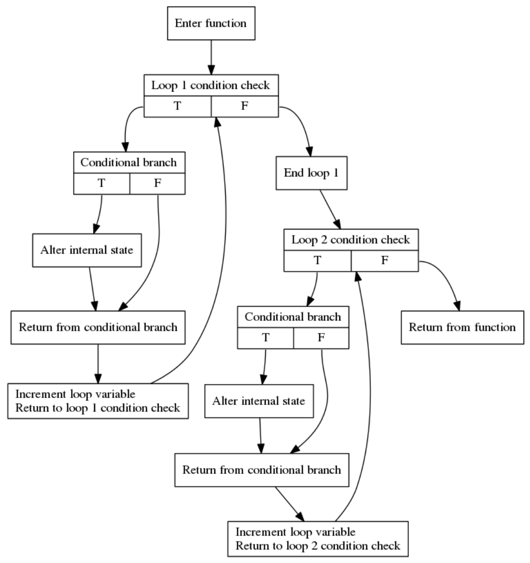

Fig. 3. CFG for double_loop program.

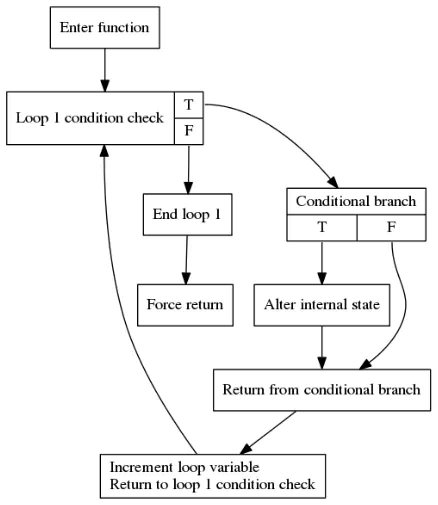

Fig. 4. CFG for double_loop program with second loop removed.

```
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;
}
```

Fig. 5. Program 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;
}
```

Fig. 6. The same program with the loop terminated after a single repetition.

IV. Developed Algorithmic Techniques

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

A. *Basis for Transformations*

We began our study by reasoning about common problems faced by symbolic execution and the structures that create these issues. Our study examines traditional symbolic execution, which uses time-limited depth-first search, as 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

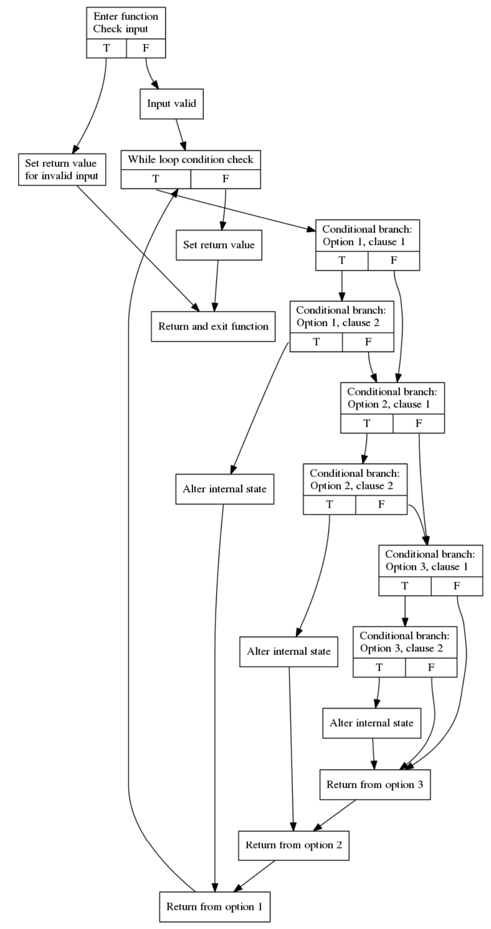
Enter function
Check input
T
F
Input valid
Set return value for invalid input
While loop condition check
T
F
Conditional branch: Option 1, clause 1
Set return value
Return and exit function
Conditional branch: Option 1, clause 2
Conditional branch: Option 2, clause 1
Alter internal state
Conditional branch: Option 2, clause 2
Conditional branch: Option 3, clause 1
Alter internal state
Conditional branch: Option 3, clause 2
Alter internal state
Return from option 3
Return from option 2
Return from option 1

Fig. 7. CFG for loop_switch program.

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.

B. Optimization Design and Implementation: The Slicer

Based off of these observations, we created several new program transformations. Each has the general goal of shrinking (or *slicing*) a program to remove parts that have a low value proposition for symbolic execution. Each "mode" of the slicer is applied to a program on a function-by-function basis. Most of them 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 highly time-intensive process,

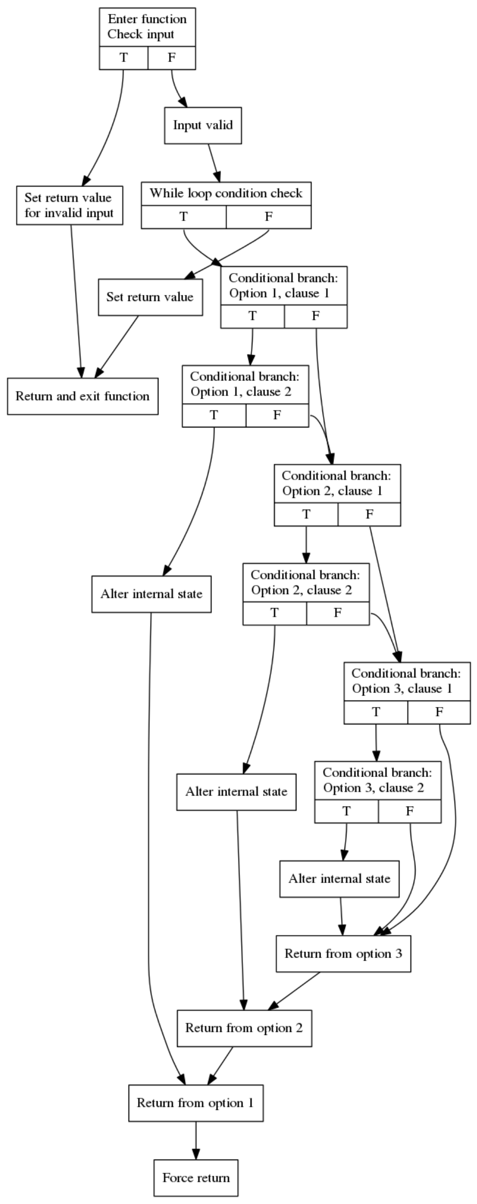
Enter function
Check input
T
F
Input valid
Set return value for invalid input
While loop condition check
T
F
Set return value
Conditional branch: Option 1, clause 1
Return and exit function
Conditional branch: Option 1, clause 2
Conditional branch: Option 2, clause 1
Alter internal state
Conditional branch: Option 2, clause 2
Conditional branch: Option 3, clause 1
Alter internal state
Conditional branch: Option 3, clause 2
Alter internal state
Return from option 3
Return from option 2
Return from option 1
Force return

Fig. 8. CFG for loop_switch with loop terminated.

and as such we included the option to limit the amount of time the slicer spends doing so on each function in the program under test. At the end of the specified time, the slicer slices the function with regard to the longest path found within that time constraint.

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 slicer 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

```
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
 }
}
```

Fig. 9. 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
}
```

Fig. 10. Pseudocode for Mode 2.

does not scale to real-world programs. While it cannot be applied to our experiment, it is important to mark down this brute-force strategy as "tried and failed."

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 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 V-D, even with an identical time budget, this mode can produce superior results.

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

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

Fig. 11. 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
}
```

Fig. 12. Pseudocode for Mode 4.

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 $ceiling(X/2) + 1$ 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. This is similar to the widely-used traditional transformation called *loop unrolling,* in this case unrolling the loop only once. While this is not a new idea, it has not been yet tested in assisting with symbolic execution.

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

V. Experimental Study

In this study we seek to answer a single core research question: If the design space for pre-symbolic execution transformations is expanded to include time-limited signature-preserving non-semantics-preserving transformations, how do these transformations impact the process of symbolic execution and the quality of the test suites thereby created?

A. 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 IV-B. Mode 0

```
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
 }
}
```

Fig. 13. Pseudocode for Mode 5.

of the slicer represents the untransformed program, and serves as our baseline for comparison.

Different programs: Unix's Coreutils are a standard group of programs used for research on symbolic execution [23], [22], [10], including the study on compiler optimizations that motivated this research. Specifically, we examined forty programs in Coreutils 6.11. We also examined six small example programs with not more than a dozen lines of C code, on which we could reasonably run KLEE to completion for proof-of-concept.

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 our design goals. Thus, we also conducted the same experiments with caching enabled to observe the interactions.

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

B. Dependent Variables

As noted in Section II, 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 transformations. We examined the following dependent variables:

Change In Line Coverage: Through gcov, a coverage reporting tool compatible with LLVM [35], 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 transformations' test suites to that of the test suite for the non-transformed version of the program, we can see how transformations 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.

C. Methodology

Each of the small 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 examined a commonly-used subset of Unix's Coreutils in our experiment; this same subset was used by Dong, Olivo, Zhang, and Khurshid [23] in their study of traditional compiler optimizations. As their work is our closest reference point, this simplifies the process of comparison; evaluation of these transformations on the rest of the Coreutils is an avenue we intend to pursue, but we here restrict our analysis to comparison against traditional semantics-preserving transformations. 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 used KLEE to run the resulting bitcode with a five-minute time limit. This time limit was also chosen to correlate with the previous work on symbolic execution and compiler optimizations, which found that increasing the symbolic execution time (to as much as 30 minutes per program) led to noticeable but overall insignificant gains in average line coverage for unaltered programs. However, as our transformed programs are in some cases completely symbolically executed within that five-minute time limit, there is a ceiling on how well they can perform. We thus do note increasing the execution time as an item for future study.

This evaluation of our four slicer modes on each of the 40 Coreutils programs selected for this research created 160 different trials. We then repeated these experiments with caching enabled.

D. Results

1) Small Examples: We tested all of the described optimizations on six small examples: two identical consecutive loops, a conditional structure inside a loop (both as described in Section III), two identical consecutive conditional statements, a single 3-way conditional branch, a simple conditional structure, and a simple function call. The results of these executions can be seen in Table I. Symbolic execution of each of these programs 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 versions tend 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 transformations 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 larger example programs, loop_switch and double_loop, it did facilitate a noticeable reduction in the number of SMT solver queries and execution time. The smaller programs saw a smaller and less consistent change in these values. Specific values for these results are omitted for brevity.

2) Unix's Coreutils: As these programs are massively more complex than our examples, the time spent in the slicer 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 II shows the average time taken for each transformation in the "Average Time" column. 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.

Fig.. 14 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 coverage percentage or number of lines covered to provide more readable results and describe the patterns generally across all programs, regardless of size. As seen in Fig. 14, the change in coverage across all programs is generally positive.

Symbolic execution times were generally reduced by the application of our transformations. The "Programs Finished Early" column in Table II shows how many programs finished before their allotted time ran out for each slicer 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 II's "Average Change in Coverage % w/Cache" column provides a breakdown of the change in coverage as compared to the non-cached version of the experiment by slicer mode. The number of queries was

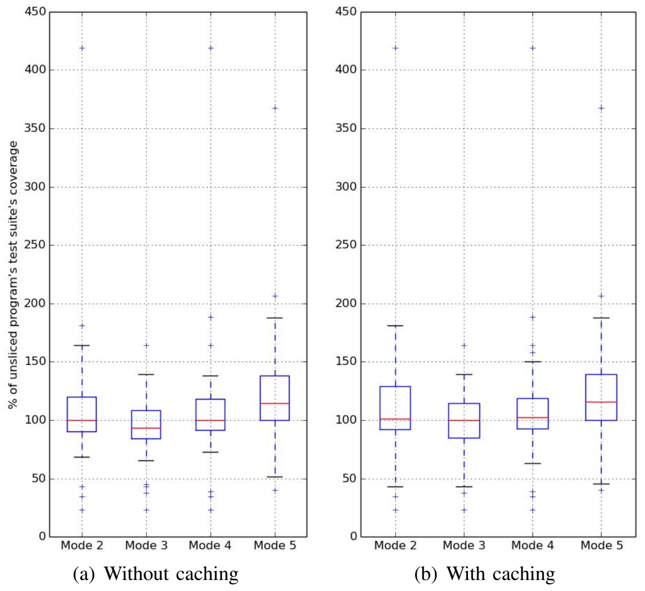

Fig. 14. Box plot for change in coverage of Coreutils programs using different slicer modes.

changed much more significantly, with all but two trials using many fewer queries. Table II's "Average Change in # Queries w/Cache" column shows these changes 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 only able to finish within their time budget when caching was enabled. The "Average % of Non-Cached Time" column of Table II shows the breakdown by slicer mode.

E. Analysis

1) Overall Performance: Across the 160 experimental trials in the non-cached version of the experiment, the transformed programs (slicer modes 2-5) produced equivalent or superior results to the originals (slicer mode 0) in 96 cases. Across the 160 trials in the cached version, the transformed programs did even better, producing equal- or higher-quality test suites in 103 cases. Table III 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 slicer 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. With these statistics in hand, we can say that our transformations outperformed traditional compiler optimizations on average in this context.

2) Caching Disabled: Fig. 14a shows a breakdown of the coverage of sliced programs' test suites as a percentage of those of unsliced programs by slicer mode. Slicer mode 5, which causes loops to terminate after one execution, was by

Program	Slicer Mode	Coverage (%)	Time (s)	Queries	Cached Coverage (%)	Cached Time (s)	Cached Queries
double_loop	0	100	324.79	1474	100	300	1205
	1	100	306.48	1106	100	300	1046
	2	100	306.93	1137	100	300	988
	3	100	305.36	1135	100	300	987
	4	100	6.19	188	100	6.38	184
	5	100	6.25	186	100	6.39	185
double_cond	0	100	71.51	1701	100	87.67	1765
	1	100	80.19	1790	100	82.77	1811
	2	100	76.65	1766	100	79.92	1728
	3	100	77.31	1779	100	80.16	1741
	4	100	77.31	1723	100	84.32	1803
	5	100	72.87	1698	100	78.72	1673
loop_switch	0	100	19.63	1619	100	18.63	1626
	1	100	1.83	123	100	1.87	125
	2	100	1.9	124	100	1.78	123
	3	100	1.88	125	100	1.8	124
	4	100	1.9	124	100	1.82	124
	5	100	3.11	196	100	2.92	193
simple_switch	0	93	3.58	208	93	3.67	215
	1	53	1.88	127	53	1.92	126
	2	93	2.2	148	93	2.29	151
	3	93	2.47	170	93	2.6	176
	4	93	2.26	148	93	2.35	150
	5	93	3.43	211	93	3.71	215
add_ints	0	100	6.07	188	100	6.28	186
	1	100	5.99	186	100	6.06	186
	2	100	6.01	186	100	6.00	184
	3	100	5.97	186	100	6.45	184
	4	100	5.86	184	100	6.27	184
	5	100	6.11	186	100	6.55	186
get_sign	0	100	2.33	160	100	2.21	162
	1	100	2.01	124	100	1.95	127
	2	100	2.05	127	100	1.97	127
	3	100	1.98	125	100	1.91	124
	4	100	1.84	123	100	1.94	123
	5	100	2.36	159	100	2.16	155

TABLE I
RESULTS FOR SMALL EXAMPLE PROGRAMS

Slicer Mode	Average Time (s)	Programs Finished Early	Average Change In Coverage % w/Cache	Average Change In # Queries w/Cache	Average % Of Non-Cached Time
Overall		78	1.7	-13845	42.12
0	0	4	0.73	-25687	32.05
2	97.7	13	2.62	-13980	45.69
3	92.6	12	2.59	-13819	53.44
4	183	18	2.13	-11973	40.20
5	98.7	21	0.55	-5534	38.66

TABLE II
METRICS FOR SLICER EXECUTION AND CHANGES WHEN ENABLING CACHING.

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 III
NUMBER OF COREUTILS IMPROVED BY EACH SLICER MODE

far the most effective of our transformations, 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 transformation, 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, transformed 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 IV 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, again as a percentage of the original program's number of queries. 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 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 transformed programs 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).

3) Caching Enabled: As referenced in Section V-D2, changes in line coverage were not large or consistent, but they were significant enough to increase the number of trials in which the transformed programs outperformed the non-transformed 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 transformed version with the addition of caching. Broadly speaking, caching made the transformations more effective; Fig. 14b 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 V. 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 VI 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.

F. 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 transformations 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 transformations, and while we believe it to be a positive one, further experimentation and exploration using different transformations, 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.

VI. Related Work

Symbolic execution [1], [2] has been studied for decades. To our knowledge, we introduce the first technique that supports symbolic execution using program transformations that are unsound *by design*.

Dong, Olivo, Zhang, and Khurshid studied the impact of standard compiler optimizations in symbolic execution [36]. The study observed that, somewhat counter-intuitively, some compiler optimizations can actually slow down symbolic execution. Cadar's recent paper [24] hypothesized about the use of non-semantics-preserving transformations for symbolic execution. Harman et al. formalized the idea of using unsound program transformations to generate test data more efficiently [25]. Our work constitutes the first application of this concept in the context of symbolic execution.

Testability transformations have been used previously to deal with issues in search-based testing, such as Baresel, Binkley, Harman, and Korel's work on the flag problem [37], [38] and McMinn, Binkley, and Harman's work on the nested predicate problem [39], [40]. Like symbolic execution, search-based testing also seeks to create test data, but searches through the input space of a program under test rather than attempting to identify concrete values using the program itself.

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 IV
AVERAGE CHANGES IN QUERIES BY SLICER MODE FOR COREUTILS, CACHE DISABLED.

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 V
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 VI
AVERAGE CHANGES IN QUERIES BY SLICER MODE FOR COREUTILS, CACHE ENABLED.

Testability transformations are used in this case to circumnavigate specific issues created by certain program structures.

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, Khurshid, and Păsăreanu [21] memoize the run of symbolic execution on the program under test as a trie and re-use it for optimizing subsequent runs. Green [14] caches the constraints that are solved during symbolic execution, in the spirit of KLEE's constraint caching [31].

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 non-determinism in the exploration to create work units and distribute the overall workload. Ranged symbolic execution [22] uses concrete inputs to split the search space in ranges, and distributes the workload by ranges.

PREfix and PREfast [41] introduce compositional analysis for symbolic execution. Recent techniques, e.g, SMART [42], SMASH [43], CompoSE [44], and others [45], [46] further developed the idea and showed its usefulness in scaling symbolic execution.

Godefroid and Luchaup [15] propose loop summarization and invariant inference as a mechanism to tackle the path explosion problem. Our work applies non-semantics preserving transformations rather than attempting to infer full loop behavior.

VII. CONCLUSION

This paper introduced non-semantics-preserving transformations designed with the goal of reducing the cost of symbolic execution for test input generation, which enables more efficient generation of higher-quality test suites than in traditional settings. We designed five new transformations based on LLVM, which were then enabled in KLEE. Upon experimentally evaluating these transformations on 40 of Unix's Coreutils, we found that we were able to achieve our goal in a majority of cases. Our transformations were able to improve coverage by up to 418%, decrease execution times and number of solver queries for the KLEE symbolic execution engine. We believe that non-semantics-preserving transformations provide a promising approach for improving the performance of symbolic execution for test generation, and hope that our work serves as a basis for developing more general techniques in this area.

ACKNOWLEDGMENTS

The authors would like to thank Mark Harman and Christian Cadar for their contributions to the foundational knowledge base of this project. This research was partially supported by National Science Foundation Grant Nos. CCF-1319688 and CNS-1239498.

REFERENCES

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

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

[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] 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.

[25] 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.

[26] N. Tillmann and J. de Halleux, *Pex–White Box Test Generation for .NET*, pp. 134–153. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[27] J. Wagner, V. Kuznetsov, and G. Candea, “-overify: Optimizing programs for fast verification,” in *14th Workshop on Hot Topics in Operating Systems*, May 2013.

[28] P. Godefroid, M. Y. Levin, and D. Molnar, “Sage: Whitebox fuzzing for security testing,” *Queue*, vol. 10, pp. 20:20–20:27, Jan. 2012.

[29] “The llvm compilation infrastructure.” llvm.org/.

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

[31] “The klee symbolic virtual machine.” klee.github.io/.

[32] “Klee coreutils study.” klee.github.io/klee/TestingCoreutils.htmls.

[33] M. Harman, A. Baresel, D. Binkley, R. Hierons, L. Hu, B. Korel, P. McMinn, and M. Roper, *Formal Methods and Testing: An Outcome of the FORTEST Network, Revised Selected Papers*, ch. Testability Transformation – Program Transformation to Improve Testability, pp. 320–344. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[34] M. Harman, “Open problems in testability transformation,” in *2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, pp. 196–209, April 2008.

[35] “The gnu coverage tool.” gcc.gnu.org/onlinedocs/gcc/Gcov.html.

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

[37] A. Baresel, D. Binkley, M. Harman, and B. Korel, “Evolutionary testing in the presence of loop-assigned flags: A testability transformation approach,” *SIGSOFT Softw. Eng. Notes*, vol. 29, pp. 108–118, July 2004.

[38] A. Baresel and H. Sthamer, “Evolutionary testing of flag conditions,” in *Proceedings of the 2003 International Conference on Genetic and Evolutionary Computation: PartII*, GECCO’03, (Berlin, Heidelberg), pp. 2442–2454, Springer-Verlag, 2003.

[39] P. McMinn, D. Binkley, and M. Harman, “Empirical evaluation of a nesting testability transformation for evolutionary testing,” *ACM Trans. Softw. Eng. Methodol.*, vol. 18, pp. 11:1–11:27, June 2009.

[40] P. McMinn, D. Binkley, and M. Harman, “Testability transformation for efficient automated test data search in the presence of nesting,” in *UK Software Testing Workshop (UK Test 2005)*, (Sheffield, UK), September 2005.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.