USING STATE MERGING AND STATE PRUNING TO ADDRESS THE PATH EXPLOSION PROBLEM FACED BY SYMBOLIC EXECUTION

THESIS

Patrick T. Copeland, Civilian

AFIT-ENG-T-14-J-3

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-T-14-J-3

USING STATE MERGING AND STATE PRUNING TO ADDRESS THE PATH EXPLOSION PROBLEM FACED BY SYMBOLIC EXECUTION

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Cyber Operations

Patrick T. Copeland, B.A.C.S.

Civilian

June 2014

AFIT-ENG-T-14-J-3

USING STATE MERGING AND STATE PRUNING TO ADDRESS THE PATH EXPLOSION PROBLEM FACED BY SYMBOLIC EXECUTION

Patrick T. Copeland, B.A.C.S.
Civilian

Approved:

/signed/	02 Jun 2014
Gilbert L. Peterson, PhD (Chairman)	Date
/signed/	30 May 2014
Maj Thomas E. Dube, PhD (Member)	Date
/signed/	30 May 2014
Barry E. Mullins, PhD (Member)	Date

AFIT-ENG-T-14-J-3

Abstract

Symbolic execution is a promising technique to discover software vulnerabilities and improve the quality of code. However, symbolic execution suffers from a path explosion problem where the number of possible paths within a program grows exponentially with respect to loops and conditionals. New techniques are needed to address the path explosion problem. This research presents a novel algorithm which combines the previously researched techniques of state merging and state pruning. A prototype of the algorithm along with a pure state merging and pure state pruning are implemented in the KLEE symbolic execution tool with the goal of increasing the code coverage. Each algorithm is tested over 66 of the GNU COREUTILS utilities. State merging combined with state pruning outperforms the unmodified version of KLEE on 53% of the COREUTILS. These results confirm that state merging with pruning has viability in addressing the path explosion problem of symbolic execution.

For my beautiful fiancée and loving parents.

Acknowledgments

I would like to thank all those who have made this work possible. A special thanks goes to my fiancée and my parents who have supported me every step of the way.

I would also like to thank my research advisor, Dr. Gilbert Peterson, for agreeing to take on this work so late in the game and providing the necessary support and guidance.

Patrick T. Copeland

Table of Contents

Page

Abstract iv

Dedication v

Acknowledgments vi

Table of Contents vii

List of Figures xi

List of Tables xii

List of Acronyms xiii

I. Introduction 1

1.1 Motivation 1
1.2 Research Objectives 2
1.2.1 Primary Objective 1 2
1.2.2 Primary Objective 2 2
1.2.3 Primary Objective 3 2
1.2.4 Primary Objective 4 2
1.3 Methodology 3
1.4 Assumptions and Limitations 3
1.5 Thesis Structure 3

II. Background 4

2.1 Symbolic Execution 4
2.1.1 Path Explosion Problem 5
2.2 KLEE 6
2.2.1 KLEE Architecture 6
2.2.2 KLEE Searcher 7
2.2.3 Measuring code coverage 7
2.3 Path Explosion Mitigation Techniques 7
2.3.1 Search Heuristics 7
2.3.1.1 User-guided 8
2.3.1.2 Depth-First Search 8

Page

2.3.1.3 Breadth-First Search 8
2.3.1.4 Best-First Search 9
2.3.1.5 Random Path Selection 9
2.3.1.6 *Generational Search* 10
2.3.1.7 Fitness-Guided 11
2.3.2 State Pruning 12
2.3.3 State Merging 12
2.3.4 Concolic Testing 14
2.3.5 Compositional Analysis 16
2.3.6 Parallel Execution 17
2.4 Summary 18

III. Methodology 20

3.1 Problem Definition 20
3.1.1 Goals and Hypothesis 20
3.1.2 Approach 21
3.2 State Merging 22
3.2.1 Dynamic State Merging 22
3.2.1.1 State Fast Forwarding 23
3.2.2 Costs and benefits of merging 25
3.3 State Pruning 25
3.4 Combining state merging and state pruning 26
3.5 System Boundaries 27
3.5.1 KLEE Flags 27
3.5.2 Reducing non-determinism 29
3.6 Workload 30
3.6.1 GNU COREUTILS 30
3.7 Performance Metrics 31
3.7.1 Code coverage 33
3.7.2 Instruction reduction 33
3.7.3 Path reduction 34
3.7.4 Average query cost 34
3.7.5 Merge time 34
3.7.6 Fast forward time 34
3.8 System Parameters 35
3.9 Factors 36
3.10 Evaluation Technique 38
3.11 Experimental Design 38
3.12 Methodology Summary 38

Page

IV. Results and Analysis 39

4.1 Overview 39
4.2 KLEE-BASE 40
4.3 Merging Results 40
4.3.1 KLEE-MERGE-5 41
4.3.1.1 Code Coverage 42
4.3.1.2 Instruction Reduction 43
4.3.1.3 Average Query Cost 43
4.3.1.4 SMT Solver Time 43
4.3.1.5 Merge and Fast-Forward Success 43
4.3.2 KLEE-MERGE-20 44
4.3.2.1 Code Coverage 44
4.3.2.2 Instruction Reduction 46
4.3.2.3 Average Query Cost 46
4.3.2.4 SMT Solver Time 46
4.3.2.5 Merge and Fast-Forward Success 46
4.3.3 KLEE-NO-FF-MERGE-5 47
4.3.3.1 Code Coverage 47
4.3.3.2 Instruction Reduction 48
4.3.3.3 Average Query Cost 49
4.3.3.4 SMT Solver Time 49
4.3.3.5 Merge Success 49
4.3.4 KLEE-NO-FF-MERGE-20 49
4.3.4.1 Code Coverage 49
4.3.4.2 Instruction Reduction 50
4.3.4.3 Average Query Cost 50
4.3.4.4 SMT Solver Time 51
4.3.4.5 Merge Success 52
4.3.5 Merging Summary 52
4.4 Pruning Results 52
4.4.1.1 Code Coverage 53
4.4.1.2 Instruction Reduction 53
4.4.1.3 Average Query Cost 54
4.4.1.4 SMT Solver Time 55
4.4.1.5 Pruned States 55
4.4.2 Pruning Summary 55
4.5 Pruning Combined with Merging Results 55
4.5.1.1 Code Coverage 55
4.5.1.2 Instruction Reduction 56
4.5.1.3 Average Query Cost 57
4.5.1.4 SMT Solver Time 58

Page

4.5.1.5 Merge and Fast-Forward Success 58
4.5.1.6 Pruned States 58
4.5.2 Pruning Combined with Merging Summary 58
4.6 Results Summary 58

V. Conclusions and Future Work 60

5.1 Contributions 60
5.1.1 Primary Contribution 60
5.1.2 Secondary Contributions 60
5.2 Limitations 61
5.3 Future Work 61

Appendix A: Code Coverage (%) for all COREUTILS 62

Appendix B: LOC and #include count for all COREUTILS 71

Bibliography 72

List of Figures

Figure Page

2.1 Example symbolic execution from Păsăreanu, et al. [1]. 5

2.2 KLEE Workflow. 6

2.3 Example program from Godefroid, et al. [2]. 10

2.4 State space tree for example program in Figure 2.3 from Godefroid, et al. [2]. . 11

2.5 Simple state merging example. 13

2.6 Example of a situation with repeated calls to the same function. 16

3.1 Merging two states together. 23

3.2 System Under Test (SUT). 29

3.3 Histogram of lines of code (LOC) for COREUTILS [3]. 32

4.1 Coverage Increases over COREUTILS for KLEE-MERGE-5. 42

4.2 Coverage Increases over COREUTILS for KLEE-MERGE-20. 45

4.3 Coverage Increases over COREUTILS for KLEE-NO-FF-MERGE-5. 48

4.4 Coverage Increases over COREUTILS for KLEE-NO-FF-MERGE-20. 51

4.5 Coverage Increases over COREUTILS for KLEE-PRUNE. 54

4.6 Coverage Increases over COREUTILS for KLEE-PRUNE-MERGE. 57

List of Tables

Table Page

2.1 Summary of Path Mitigation Techniques. 19

3.1 List of GNU COREUTILS tested. 31

3.2 Experimental Factors. 37

4.1 Summary of Coverage Results for All Algorithms. 40

4.2 Impact of KLEE-MERGE-5 on Average Code Coverage, Instructions, Queries, Query Constructs, Average Query Cost, Successful Merges, and Successful Fast-Forward. 41

4.3 Impact of KLEE-MERGE-20 on Average Code Coverage, Instructions, Queries, Query Constructs, Average Query Cost, Successful Merges, and Successful Fast-Forward. 44

4.4 Impact of KLEE-NO-FF-MERGE-5 on Average Code Coverage, Instructions, Queries, Query Constructs, Average Query Cost, SMT Solver Time and Successful Merges. 47

4.5 Impact of KLEE-NO-FF-MERGE-20 on Average Code Coverage, Instructions, Queries, Query Constructs, Average Query Cost, SMT Solver Time and Successful Merges. 50

4.6 Impact of KLEE-PRUNE on Average Code Coverage, Instructions, Queries, Query Constructs, Average Query Cost, SMT Solver Time and Pruned States. . 53

4.7 Impact of KLEE-PRUNE-MERGE on Average Code Coverage, Instructions, Queries, Query Constructs, Average Query Cost, SMT Solver Time, Successful Merges, Successful Fast-Forward and Pruned States. 56

List of Acronyms

Acronym	Definition
ASLR	address space layout randomization
CFG	control flow graph
SMT	Satisfiability Modulo Theories
NURS	non-uniform random search
DFS	depth-first search
BFS	Breadth-first search
SPD	Symbolic Program Decomposition
LLVM	low-level virtual machine
SUT	system under test
CUT	component under test
CFG	control flow graph
KLOC	thousands (kilo) of lines of code
LOC	lines of code

USING STATE MERGING AND STATE PRUNING TO ADDRESS THE PATH EXPLOSION PROBLEM FACED BY SYMBOLIC EXECUTION

I. Introduction

1.1 Motivation

Distribution of computer software with latent security vulnerabilities sadly remains the norm throughout the industry. The recently disclosed "Heartbleed" vulnerability in the OpenSSL library is a prime example of vulnerabilities having wide ranging consequences [4]. Software with vulnerabilities ultimately threatens developers' reputations and costs users and developers time and money. Thus, the need for improved software testing methods is urgent and compelling. Recent efforts in this area automate program analysis techniques using model checking and symbolic execution [2, 5–7]. These methods often find subtle software bugs missed by other techniques. Yet, despite their success, these methods suffer from scalability issues, one of which is the so-called path explosion problem. That is, as the complexity of the program grows, the number of possible paths within the program grows too large for effective analysis.

The primary challenge to scaling symbolic execution techniques is the number of possible paths within the program. The number of possible paths through a program is a function of the conditionals and loops contained within the program. Consider a simple `if-else` control structure. If the given condition is `true`, the symbolic execution tool executes the `if-block`. This represents one possible path within the program. The `else-block` represents a second path. Each additional `if-else` statement adds additional paths. In general, the number of paths in a program grows exponentially in relation to the number of conditional statements. A naïve, exhaustive approach to the path explosion problem is infeasible for any nontrivial program. Applications often contain millions of lines of code. Successful symbolic execution tools must address path explosion, and several recent techniques show promising results.

1.2 Research Objectives

1.2.1 Primary Objective 1.

Provide a thorough analysis of the effects of state merging on symbolic execution. State merging has the potential to save the symbolic execution tool a great deal of work by reducing the occurrence of redundant path exploration. However, the result of two merged states is a single, child state where the path constraint is a disjunction of the two parent states' path constraints, which creates additional strain on the Satisfiability Modulo Theories (SMT) solver.

1.2.2 Primary Objective 2.

Provide a thorough analysis of the effects of state pruning on symbolic execution. State pruning is an additional method of reducing the state space during symbolic execution. The reduction of duplicate states can lead to an increase in overall coverage.

1.2.3 Primary Objective 3.

Introduce a novel algorithm that combines the ideas of state merging and state pruning.

This research introduces a novel state reduction algorithm to address the path explosion problem of symbolic execution. The new algorithm combines the previously researched techniques state merging and state pruning.

1.2.4 Primary Objective 4.

Provide a thorough analysis of the effects of the novel state merging and pruning algorithm.

This research implements a prototype of the state merging and state pruning algorithm in the KLEE symbolic execution tool [7]. The prototype is tested on 66 of the GNU

COREUTILS [3], a widely used suite of utilities in Unix/Linux systems. A performance analysis of the prototype is provided focusing on the changes in code coverage with respect to the unmodified version of KLEE.

1.3 Methodology

This research attempts to identify an effective method to mitigate the current path explosion problem faced by the symbolic execution tools. The system under test (SUT) is the KLEE program analysis tool and the component under test (CUT) is the path explosion mitigation technique. The techniques tested are state merging and state pruning. A prototype of each algorithm is implemented with the KLEE symbolic execution tool. The effectiveness of the each algorithm is measured in terms of code coverage, SMT solver time, average query cost, merge time, and fast forward time.

1.4 Assumptions and Limitations

This research does not use KLEE in the optimal configuration with respect to code coverage. The unmodified version of KLEE used to compare the path mitigation algorithms uses the built-in coverage oriented search. The optimal configuration is the coverage search interleaved with a random path search. The random path search is omitted to reduce non-determinism and allow for a more direct comparison between the base version of KLEE and the algorithm prototypes.

1.5 Thesis Structure

This thesis follows the below structure. Chapter 2 provides the necessary background information, as well as a review of the current literature relating to symbolic execution. Chapter 3 describes the algorithm that this work evaluates, describes the main factors used to modify the algorithm and the metrics used to evaluate performance. Chapter 4 is a thorough analysis of the resulting data from the algorithm. Chapter 5 summarizes the contributions of this work and provides suggestions for future work.

II. Background

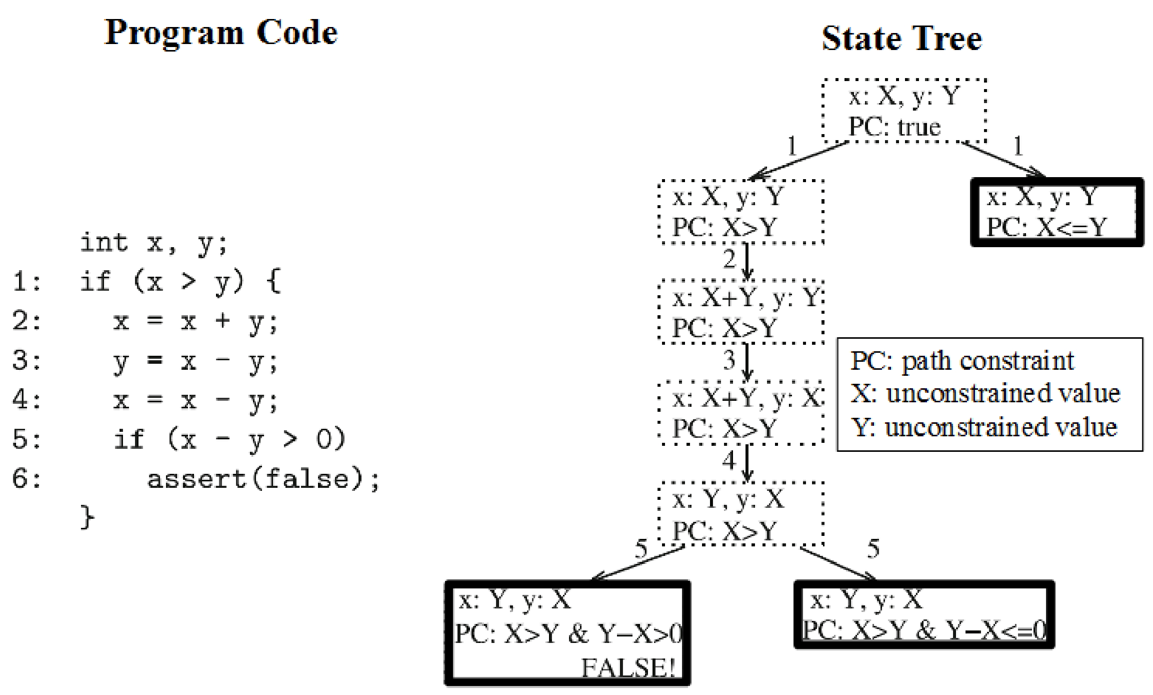
This work introduces a novel algorithm which combines state merging and state pruning to achieve an increase in code coverage. The testing is done on the GNU COREUTILS [3].

This chapter provides the necessary background information regarding symbolic execution and the KLEE symbolic execution tool. The chapter concludes with a review of current techniques to address the path explosion problem of symbolic execution.

2.1 Symbolic Execution

Symbolic execution has been proposed as an effective way to find software vulnerabilities within a program [2, 5–7]. Symbolic execution evaluates a program using symbolic values as substitutes for the actual, concrete values. As the program executes, symbolic expressions represent values within the program. The symbolic execution tool must maintain program state information for each active state during exploration. This includes call stack, memory, registers, and all symbolic values. For each active state, the tool must maintain a path constraint. A path constraint is a first order, quantifier free formula over symbolic expressions [8]. Figure 2.1 provides an example execution. When the program encounters a branching statement with symbolic data, the tool generates new constraints. The symbolic execution tool conjuncts the new constraints with the current path constraint. This can be seen in the transition from State 1 to State 5. For a given path to remain feasible, the path constraint must be satisfiable. That is, for a given formula to be satisfiable, there must be a binding of true and false values to the variables in the formula which result in the entire formula being true.

Stepping through Figure 2.1, each box corresponds to a statement within the program. The variables x, y begin as unconstrained values X, Y. As execution continues, the values



```
    int x, y;
1:  if (x > y) {
2:    x = x + y;
3:    y = x - y;
4:    x = x - y;
5:    if (x - y > 0)
6:      assert(false);
    }
```

Figure 2.1: Example symbolic execution from Păsăreanu, et al. [1].

are updated, as well as the path constraint. Notice that the conditional statements on line 1 and line 5 create a new branch in the execution tree. Each branch statement generates a new condition within the path constraint that must be satisfied for exploration of the path to continue. The path constraint generated is then passed to the SMT solver to determine if a binding of values exist to make the expression true. After line 5, since $X > Y$ and $Y - X > 0$ cannot both be true at the same time, no binding of values exist to make the left branch of the tree's path constraint satisfiable. Thus, the branch is unsatisfiable and exploration of that branch halts.

2.1.1 Path Explosion Problem.

When exploring the entire state space of a real-world application, a symbolic execution tools face a path explosion problem. In general, the number of paths within a program grows exponentially with respect to conditionals and loops [9]. A naïve approach to searching the state space will fail to achieve high code coverage due to the exponential growth of paths.

2.2 KLEE

KLEE is a symbolic execution tool designed to work with C source files [7, 10]. KLEE has been used to perform a variety of tasks including high code coverage of real-world applications and bug finding. The original work was tested on the GNU COREUTILS and BUSY-BOX utility suites.

2.2.1 KLEE Architecture.

To use KLEE on C source files, the files must first be compiled into low-level virtual machine (LLVM) bitcode. KLEE acts as an interpreter for the LLVM bitcode. To produce LLVM bitcode, the user must compile C source files using either the `llvm-gcc` [11] compiler or `clang` [12] compiler. Figure 2.2 presents a sample KLEE workflow.

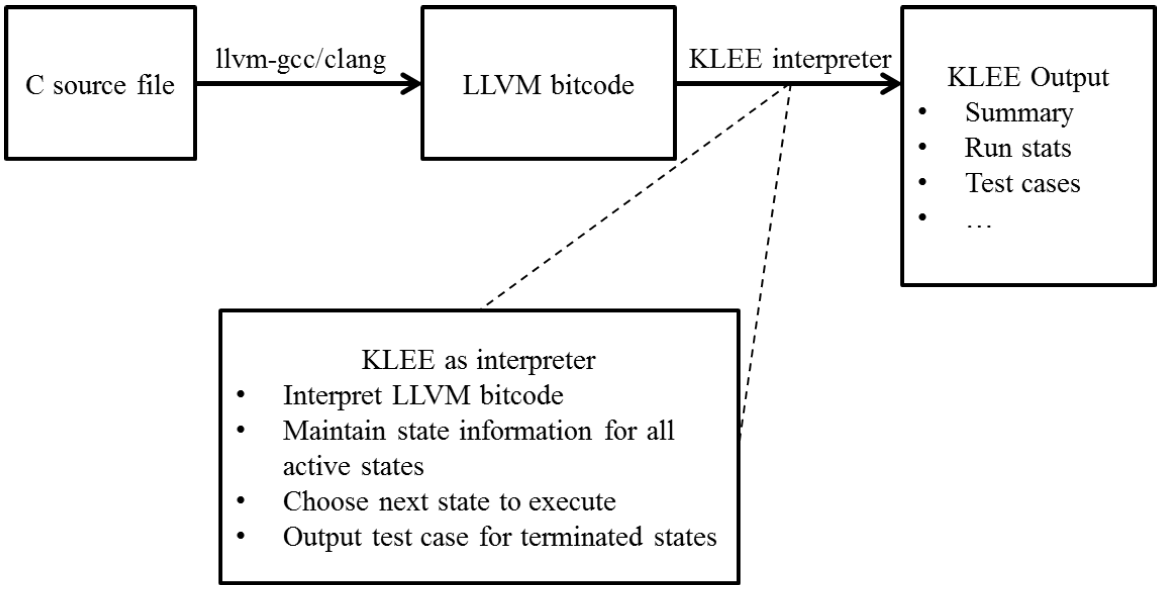


Figure 2.2: KLEE Workflow.

2.2.2 KLEE Searcher.

The KLEE Searcher class handles the selection of the next state to execute during exploration of the state space. KLEE's modular design allows KLEE to implement different search strategies. The base version of KLEE includes the following search strategies: random path, depth-first search (DFS), and other weighted heuristics geared towards code coverage, depth, query cost and instruction count [13].

2.2.3 Measuring code coverage.

For each path that is explored by KLEE, the tool writes out a test file as a `.ktest` with values that represent a path. Included with KLEE is a command line utility, `klee-replay` that takes a compiled C program and KLEE test files and executes the program with the given input. An external test coverage tool is required to gather actual code coverage. The original KLEE work uses the GNU `gcov` utility [14]. After executing a specific utility with each test case generated by KLEE, `gcov` provides a percentage of lines of code executed.

2.3 Path Explosion Mitigation Techniques

An effective symbolic execution tool for real world applications must address the path explosion problem. This section describes how current tools overcome the path explosion problem and categorizes the tools into the following classes: search heuristics [2, 5, 6, 9, 15–17], state merging [18–20], state pruning [21], concolic testing [2, 5, 6, 16, 22, 23], compositional analysis [24–26], and parallel execution [27].

2.3.1 Search Heuristics.

Search heuristics use knowledge about a given state to make an informed decision about the next appropriate action. Heuristics-based searches are common in artificial intelligence applications where the search space is too large for an exhaustive search. A search heuristic often tries to maximize a given value. For symbolic execution, path coverage or exploration of interesting paths are two values a tool may try to maximize.

2.3.1.1 User-guided.

A user-guided search heuristic uses the skill of a human being to guide the execution of a program. With a skilled operator, a user-guided approach can be effective at picking interesting paths that could possibly lead to a vulnerability. However, the lack of automation and the required reverse code engineering skill of a human places a limit on this approach.

A tool which implements the user-guided search is Jiseki [9]. Jiseki is a bounded model checking tool for x86 binary programs developed at the Air Force Institute of Technology (AFIT). Jiseki creates a bit-vector logic model based on x86 binaries and uses the model to reason about the given program. The operator uses a GUI-based plug-in with IDA, a disassembly and debugging tool, to view the exploration of the state space and guide the program down interesting paths.

2.3.1.2 Depth-First Search.

Depth-first search (DFS) is a straightforward search strategy that has many applications across different domains. DFS will continue down the same path until reaching a terminal state. After reaching a terminal state, DFS recurses up the search tree and explores a new branch. Since DFS will explore the same path until termination, the algorithm excels at exploring a given path neighborhood but does poorly in overall path coverage. Many of the older symbolic execution tools use a depth-first search [5, 16].

2.3.1.3 Breadth-First Search.

Breadth-first search (BFS) is an additional search strategy used in many different domains. BFS attempts to explore all states at a given depth before moving deeper into the search tree. In the symbolic execution domain, this provides an exhaustive exploration of shallow paths. However, bugs that occur deep within the search tree will be missed. KLEE [7] includes functionality for BFS.

2.3.1.4 Best-First Search.

Best-first search looks at the possible children of a given state and chooses the child node which maximizes a specified value. Depending on the goal of the algorithm, “best” can have many different meanings. For symbolic execution, the maximized value could be the likelihood that a child state will lead to a bug. Maximizing a value that represents “most likely to lead to a bug” is similar to the what a reverse code engineer does. The human operator runs a mental best-first search using prior reverse code knowledge to identify certain calls that have the potential to induce vulnerabilities. Developing an algorithm to capture this human intuition is difficult.

Alternatively, the best child state could be the least explored child. By placing a high value on unexplored paths, the tool is trying to increase overall code coverage by forcing exploration down neglected paths. EXE [6], a symbolic execution tool developed at Stanford University, implements a best-first search, which defines “best” as those unexplored paths. EXE demonstrates an improvement over traditional fuzzing tools.

KLEE [7] implements a suite of best-first searches as non-uniform random search (NURS). Instead of a uniform distribution, NURS are weighted towards a specific goal. The options for goals include: code coverage, depth, minimum distance to uncovered instruction, and a query cost estimate. KLEE’s highest performing configuration with respect to code coverage is the search strategy that switches between a random-path search and a best-first search maximizing code coverage. This round-robin switching prevents the possibility of one search strategy completely dominating and getting “stuck” in a certain area of code.

2.3.1.5 Random Path Selection.

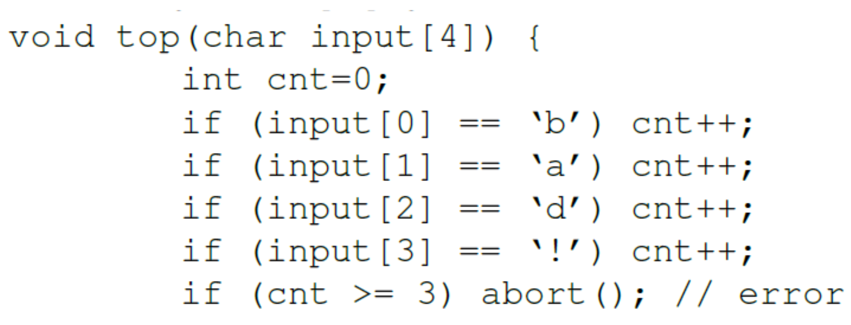
A uniform random path search policy gives each state an equal chance of being selected next. While this technique is not sufficient by itself for real world applications, the combination of the random search and an additional search strategy can be beneficial.

Cadar, et al. [7] combine a random search strategy with a coverage oriented strategy. The advantage of including a random search is that the random search avoids getting stuck in small segments of code. While a best-first search may heavily favor a small subset of the execution tree, the random search forces different sections of the tree to be explored with equal probability.

2.3.1.6 Generational Search.

Generational search is a strategy designed specifically for symbolic execution by Godefroid, et al. [2]. To ensure better code coverage, generational search systematically negates each constraint within a given path constraint. By negating each constraint, the algorithm explores execution down as many different paths as possible.

SAGE (*S*calable, *A*utomated, *G*uided *E*xecution) [2, 28] is a symbolic execution tool developed by Microsoft, which implements a generational search algorithm. SAGE is regularly used to test Microsoft applications and succeeds in finding many bugs missed by other testing methods [28]. SAGE also makes use of an additional technique, concolic testing, discussed later in this section.



```
void top(char input[4]) {
        int cnt=0;
        if (input[0] == ‘b’) cnt++;
        if (input[1] == ‘a’) cnt++;
        if (input[2] == ‘d’) cnt++;
        if (input[3] == ‘!’) cnt++;
        if (cnt >= 3) abort(); // error
```

Figure 2.3: Example program from Godefroid, et al. [2].

Consider the example in Figure 2.3 and Figure 2.4. The leftmost path is the first path explored. To force execution down a separate path, the generational search negates

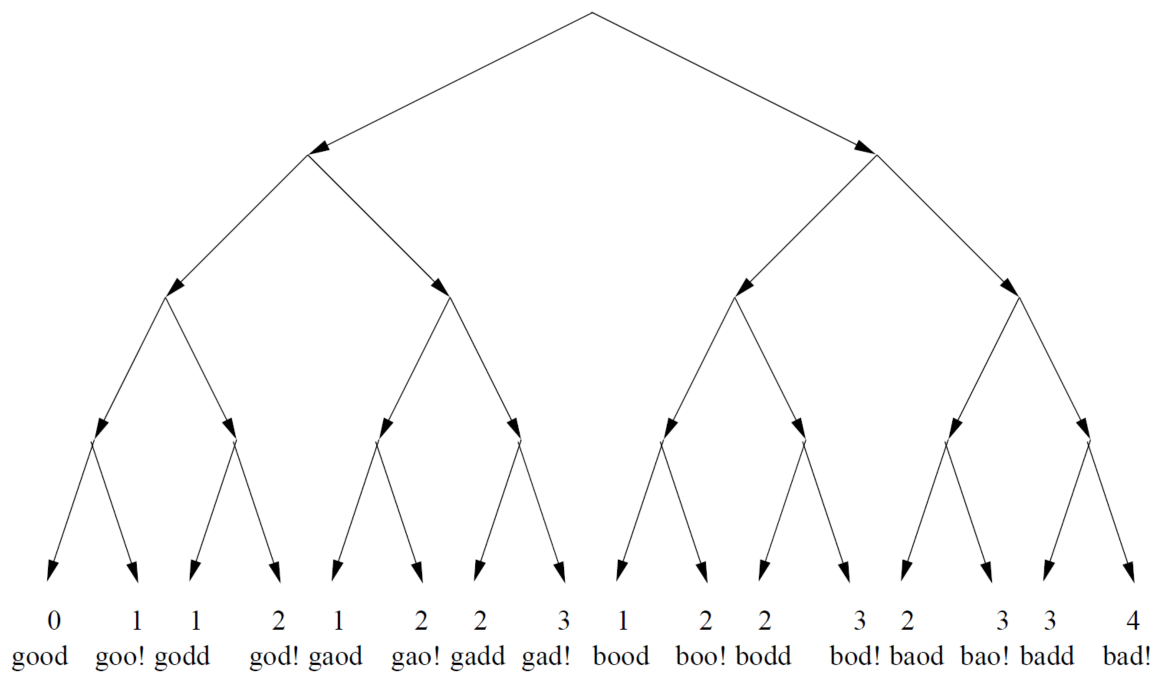


Figure 2.4: State space tree for example program in Figure 2.3 from Godefroid, et al. [2].

the final constraint, `input[3] == '!'`. The next execution will satisfy the `input[3] == '!'` condition and explore a new path. The generational search will explore all possible expressions of the path constraint.

2.3.1.7 Fitness-Guided.

Pex [23] is an additional symbolic execution tool designed by Microsoft specifically for .NET applications. To improve the performance of Pex, Xie, et al. [15] developed a new search strategy, Fitnex. Fitnex is a fitness-guided search strategy that computes *fitness values* that are used to guide the execution of the program.

Fitness-based searches attempt to recreate biological phenomenons, such as natural selection and mutation. Fitnex is similar to a best-first search where the algorithm chooses a target (primarily exploring new paths) and computes a fitness value for each path. The fitness values represents how close the given path is to reaching the desired target. Test results comparing Pex with and without Fitnex show a four times improvement in code coverage when using Fitnex.

2.3.2 State Pruning.

In traditional symbolic execution, when a given path splits into two distinct paths, evaluation of those paths occurs separately until termination. State pruning aims to take advantage of the fact that two separate paths could be equivalent. Showing equivalence of two states allows the tool to prune one of the states, which prevents duplicate work.

The difficulty with state pruning is determining equivalence. In most cases, proving complete equivalence is too computationally expensive to reap any benefit. Therefore, an approximation of equivalence is made. When two states meet a closeness threshold, the symbolic execution tool combines the threads of execution. By estimating equivalence, the tool may under- or over-approximate the system depending on joining procedures. That is, by not exploring both paths, the tool may fail to capture all of the program's behavior or introduce a capability that does not exist within the program. Any under-approximation may lead to bugs being missed, while an over-approximation could lead to false positives.

Bugrara and Engler [21] introduce a redundant state detector algorithm and build a prototype in KLEE. The focus of Bugrara and Engler's work is high-code coverage testing. As such, they decide that two states are equivalent if they will execute the same lines of code. On 55 of the 66 COREUTILS tested, Bugrara and Engler's work achieved at least the same code coverage as the base version of KLEE.

2.3.3 State Merging.

As with state pruning, state merging attempts to reduce redundancy in the exploration of the state space. If two states happen to arrive at the same program counter, there is the potential of merging the two states into one. This is done by disjuncting the two path constraints.

Figure 2.5 demonstrates that when two states are at the same program point, S_3 and S_4, combining the two states, there is the potential for pruning a significant amount of the search space.

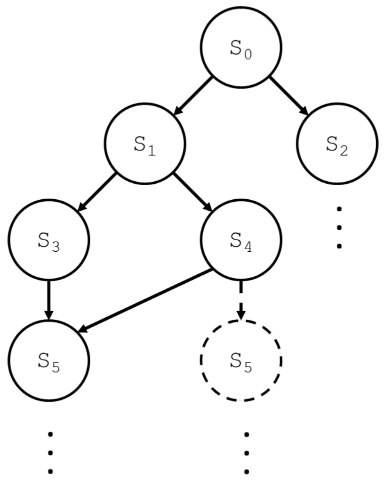


Figure 2.5: Simple state merging example.

It is sometimes the case that the additional complexity of the disjuncted path constraint may be more computationally expensive than exploring each states separately.

Recent work [18, 19] explores the tradeoff between the additional overhead of state merging and consequences of approximation with the benefit of reduced paths. Work by Hansen, et al. [18] provides mixed results, giving an improvement on a small subset of programs. As the system becomes more complex, traditional symbolic execution overtakes the state merging method.

RWSet [19] is an additional algorithm that implement state merging. Boonstoppel, et al. describe the motivating ideas behind RWSet as:

1. two states are equivalent if they produce the same *effects*, and

2. values that differ between states that are never read in subsequent states can be ignored.

To implement this approach, RWSet maintains a *cache* of visited states. Whenever there is a *cache hit* [19], RWSet records the path constraint and generates an input to reach the halting state. The most important feature of RWSet is a method to determine "live variables." When trying to decide if a variable is still alive, RWSet does a DFS from the given state and records any calls to read the variable. RWSet is implemented in conjunction with the concolic testing tool, EXE [6] (discussed in the following section) and has shown significant reduction in time required to reach similar branch coverage as EXE without RWSet.

Kuznetsov, et al. [20] represents the most recent work on state merging. Kuznetsov, et al. provide a novel approach to estimate the cost of merging two states, *query cost estimation*. Two states merge only when the estimated cost of doing so is less than the expected gain. By adding the cost metric, the Kuznetsov, et al. method demonstrates orders of magnitude speedup over alternative methods.

2.3.4 Concolic Testing.

Traditional symbolic execution relies solely on symbolic values when examining a system. Strictly using symbolic values can be cumbersome when dealing with complex data structures and pointers. Additionally, path constraints can grow very large, which slows the work of the SMT solver. A proposed alternative to pure symbolic execution is *concolic testing*. The concolic approach combines symbolic execution with actual execution of the code, *concrete execution*. The motivation for using a mixed approach is to:

- reduce the complexity of reasoning about complex data structures and pointers,
- take advantage of the speed of concrete execution, which does not rely upon an SMT solver, and

- use symbolic execution to guide execution of the program.

As the code is executed concretely, the tool generates symbolic constraints which are used to create a new concrete input into the program to increase the chances that execution on the input will force exploration down a different path.

Because of the difficulty of symbolically representing complex data structures, many of the early symbolic execution tools use concolic testing. These tools include DART [5], CUTE [16], EXE [6], PEX [23], and SAGE [2]. Test results show a significant increase in path depth and an improvement of path coverage over strict symbolic execution.

The SAGE example shown in Figures 2.3 and 2.4 demonstrate the concolic execution process. The string "good" is *concrete* input given to the function. As the program executes on the string "good", SAGE maintains the path constraint of *symbolic* values based on the four conditional statements. To generate a new concrete input, SAGE negates the last constraint and feeds the new concrete input "goo!". SAGE continues systematically negating each constraint in the path constraint, generating new concrete inputs to guide execution down different paths.

Majumdar, et al. [22] provide a variation on concolic testing where the algorithm switches between two modes: random fuzzing and concolic testing. The hybrid concolic algorithm operates in a fuzzing mode until a given area of the tree is well explored. When the algorithm senses that fuzz testing is no longer exploring new paths, the algorithm switches to concolic testing mode and forces execution of new paths. The motivation of the hybrid approach is taking advantage of the speed of generating random inputs, and a strength of fuzz testing, which is the ability to explore a given neighborhood of the state tree. A weakness of fuzz testing is the tendency to get stuck re-exploring a small subset of the state space. By adding concolic execution, the search forces execution down a new path. Majumdar, et al. implemented the hybrid approach on top of CUTE, finding a 4 times increase in code coverage over pure fuzz testing and 2 times increase in code coverage over concolic testing.

2.3.5 Compositional Analysis.

Compositional analysis attempts to alleviate the path explosion problem by breaking a large program into smaller, logical pieces (i.e. functions) and reasoning about pieces individually. After reasoning about a function, the tool generates a summary of the function that each subsequent call utilizes. By using the function summary, instead of stepping through the function's state space, large branches of the overall state space can be pruned.

To see the potential savings, consider the code snippet in Figure 2.6. Additionally, assume there are `n` different paths through the function `foo`. The program could potentially call the function `foo` a `bound` number of times, resulting in $n \times bound$ possible paths through the program. The preferred method is executing `foo` once and capturing the effects of the function call in a function summary. In subsequent calls to `foo`, the function summary would be used and exploration of `foo` would be unnecessary, reducing the amount of redundant work.

```
while (i < bound) {
        foo();
}
```

Figure 2.6: Example of a situation with repeated calls to the same function.

A difficulty with this approach is computing the function summary which captures all the functions behavior. As with computing equivalence of states, an approximation is required. Depending upon how the approximation is made, the function summary could under- or over-estimate the system's functionality.

Godefroid, et al. [24] explore the compositional approach in the SMART (Systematic Modular Automated Random Testing) algorithm. SMART works by testing a function and expressing the function summary as precondition inputs to the function and postcondition outputs of the function. Godefroid, et al. prove the correctness of SMART with respect to DART. That is, any path explored by the DART algorithm will also be explored by the SMART algorithm using composition. The primary advantage of SMART is the scalability to large applications.

Santelices, et al. [26] introduce a technique called Symbolic Program Decomposition (SPD). SPD utilizes composition based on *path families* instead of functions. A path family is "a group of paths that share common control dependencies" [26]. The approach of SPD is similar to a state merging approach. SPD attempts to capture the behavior as of a path family as a whole.

SPD also contains unique features that allows the algorithm to scale to large applications. As SPD executes, the algorithm allows the "dropping" of constraints, resulting in an under-approximation in the path constraint which corresponds to an over-approximation of the entire system. Decreasing the number of constraints in the path constraint reduces the amount of work required from the SMT solver, which improves overall performance.

2.3.6 Parallel Execution.

The final approach receiving attention from the research community is a parallel implementation of symbolic execution strategy. In general, more workers are able to accomplish more work. The path explosion is a large problem that could benefit from more workers. Parallel symbolic execution aims to take advantage of increases in cheap and powerful computational tools.

In order to maximize the benefit of many workers concurrently using symbolic execution to examine a program, the amount of duplicate work must be minimized. That is, multiple workers exploring the same path does not provide any improvement over a single worker. The difficulty of parallelizing symbolic execution is the lack of:

- an effective method to partition the state space, and
- a system for worker processes to communicate during execution.

Staats, et al. [27] propose a parallel approach to symbolic execution called Simple Static Partitioning. The method partitions the state space using predetermined conditions. The conditions are generated by first performing a round of symbolic execution which gathers initial path conditions. Simple static partitioning takes the pre-conditions and partitions the state to eliminate overlap of the conditions. After partitioning the state, the algorithm starts the worker processes using the Java Pathfinder Framework (JPF) [29], which includes a symbolic execution plugin, to explore the partitions.

The framework proposed by Staats, et al. does not require explicit communication between the worker nodes. The approach does allow for communication via remote listener workers. The listener workers maintain a "cache" of explored paths and are able to signal a worker process to terminate execution of previously explored path. The results show a decrease in the amount of time to reach similar path coverage as the JPF [29].

2.4 Summary

The current techniques to address the path explosion problem include search heuristics, state merging, state pruning, concolic, compositional analysis, and parallel execution. Table 2.1 groups the current research efforts into their respective categories. The next chapter presents a novel algorithm that combines the state merging and state pruning techniques together.

Table 2.1: Summary of Path Mitigation Techniques.

Strategies	**Examples**
Search Heuristic	[2, 5, 6, 9, 15–17]
State Merging	[18–20]
State Pruning	[21]
Concolic	[2, 5, 6, 16, 22, 23]
Compositional Analysis	[24–26]
Parallel Execution	[27]

III. Methodology

The program analysis of symbolic execution suffers from a path explosion technique. This research attempts to identify effective means of overcoming this problem and increase overall code coverage of tested software.

This chapter provides the problem definition as well as the goals and hypotheses. A description of the algorithms under test is given, along with the experimental design for this research.

3.1 Problem Definition

KLEE [7] is a program analysis tool that uses symbolic execution to test C code. KLEE is chosen for this research because it is an open-source project with an active developer community. Also, KLEE has been used in previous symbolic execution research. As with other symbolic execution tools, KLEE suffers from a path explosion problem. That is, the number of possible paths within a program grows extremely large due to the number of loops and conditionals contained within the program. To efficiently analyze real applications, an improved method of addressing the path explosion problem is needed. Adopting the notation of Bugrara and Engler [21], this research refers to the unmodified version of KLEE as KLEE-BASE.

3.1.1 Goals and Hypothesis.

The goal of this research effort is to measure and compare the effectiveness of the novel algorithm state merging combined with state pruning. To help assess the effectiveness state merging and state pruning are also tested separately.

State merging attempts to reduce the amount of duplicate work done by the symbolic execution tool. If two states arrive at the same program point, the path constraint for each parent state is disjuncted together, resulting in a single child state. This allows the exploration of a single, more complex state.

Like state merging, state pruning reduces the amount of duplicate work by exploiting the fact that two separate paths may be identical. That is, if execution of a state branches into two separate states at point A and those states converges at point B, the instructions executed between point A and point B may not change the states with respect to the symbolic expression. In the case where the two resulting states are equivalent, only one of the paths must be explored further.

State merging combined with state pruning will attempt to apply both of the reductions from each individual algorithm. During exploration of the state space, the algorithm looks for opportunities to both merge and prune. The two algorithms complement one another since they both can only occur when two states arrive at the same program point.

It is hypothesized that each techniques will improve the performance of the KLEE tool by reducing the amount of duplicate work required to analyze a given program. The reduced work should allow deeper exploration of paths, as well as exploration of new paths that would not be reached by the base program in the same amount of time. This additional exploration will increase the overall code coverage. The addition of each technique introduces overhead not required by the unmodified KLEE tool, but because of the large number of paths within a program, this overhead is expected to be overcome by the efficiency of the technique.

3.1.2 Approach.

The approach to achieve the above goals is to compare the performance of the symbolic execution tool while analyzing the GNU COREUTILS [3]. To perform testing, both the state merging and state pruning are implemented as instances of the KLEE Searcher class. The different symbolic execution configurations are KLEE-BASE, KLEE with the addition of a state merging (KLEE-MERGE), KLEE with the addition of a state pruning (KLEE-PRUNE), and KLEE with the additions of state pruning and state merging (KLEE-PRUNE-MERGE).

3.2 State Merging

State merging attempts to address the path explosion problem by reducing the number of states that must be explored by combining states that arrive at the same program point into a single, more complex state. States are merged by disjuncting the path constraints for each state. Figure 3.1 illustrates this concept. Notice, the path constraint in the final state is a disjunction of the two previous path constraints.

The state merging strategy is referred to as KLEE-MERGE. In the context where a maximum number of merges is associated with state merging, the strategy is referred to as KLEE-MERGE-⟨max merges⟩. When fast-forwarding functionality (discussed in Section 3.2.1.1) is disabled, the state merging strategy is referred to as KLEE-NO-FF-MERGE. Again, in the context where a maximum number of merges is associated with state merging, the strategy with no fast-forwarding is referred to as KLEE-MERGE-⟨max merges⟩.

Algorithm 3.1 provides state merging in algorithmic notation.

3.2.1 Dynamic State Merging.

State merging can be done either statically or dynamically. Static state merging requires the building and traversal of a control flow graph (CFG) [20]. When traversing the CFG, the algorithm identifies all join points and attempts to merge at all join points. Part of the issue with using a CFG is that it may model behavior that is not possible for the program to execute. This research chooses to avoid the preprocessing steps of building and searching a CFG by using a dynamic approach.

Kuznetsov, et al. [20] propose the idea of dynamic state merging as an alternative to static state merging. Dynamic state merging requires an arbitrary, underlying search

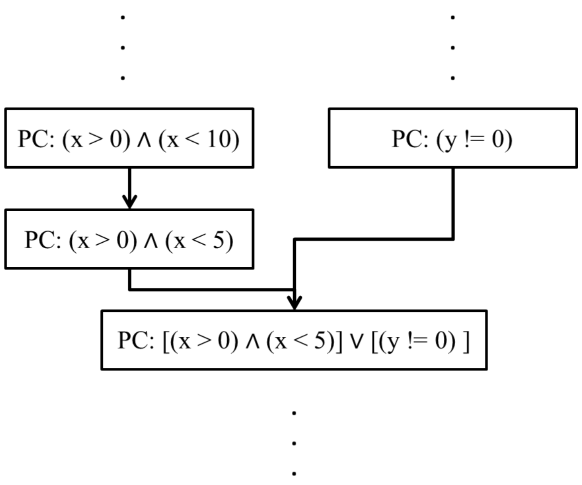


Figure 3.1: Merging two states together.

strategy that implements a `pickNextState` function. While any strategy can be used, some are less effective at achieving the goals of state merging. For example, a DFS will follow a single path until termination, removing the possibility of merging. This research uses the underlying best-first search maximizing code coverage search included with KLEE.

3.2.1.1 State Fast Forwarding.

States must be at the same program point in order to merge. Kuznetsov, et al. [20] introduce the idea of state *fast-forwarding*. With state fast-forwarding, the searcher maintains a history of previous program states. A state is a candidate for fast-forwarding if the state could have been merged with a state in the history of another active state. Kuznetsov, et al. propose that two states that could have merged in the past will likely be able to merge in the near future. The state is given priority and stepped forward a bounded number of times. Not all candidates for fast-forward will successfully merge. It is possible that the fast-forward state branches in a different direction, diverging from the potential merge candidate.

This research adopts a similar strategy for fast-forwarding; however, instead of maintaining a history of execution states and checking whether a state is a candidate to merge based on another state's history, this research takes a more simplified approach and maintains a history of program points. The idea being that if two states executed the same instruction, they are likely in the same area of the search space and this distance between two states is sufficiently small to allow for fast forwarding. Notice that by maintaining only program points, there is no way of knowing if past states could have merged. Instead this fast-forward approach only attempts to force states to the same program point.

Algorithm 3.1 State Merging - selectState()

```
Require: baseSearcher
 1: // fastForwardState - state selected to be fast-forwarded
 2:
 3: if $fastForwardCount > 0$ then
 4:     $state \leftarrow fastForwardState$
 5: else
 6:     $state \leftarrow$ baseSearcher.selectState()
 7: end if
 8: for all states $s$ at $state.inst$ do
 9:     // Returns true if $state$ and $s$ can merge
10:     if canMerge($state$, $s$) then
11:         // Merges s into state
12:         doMerge(state, s)
13:         // Disable fast-forward mode
14:         $fastForwardMode \leftarrow 0$
15:         Delete $s$
16:         return $state$
17:     end if
18: end for
19: if $fastForwardCount \leq 0$ then
20:     // Returns true if $state$ is found in the history of another active state
21:     if canFastForward($state$) then
22:         $fastForwardCount \leftarrow 10$
23:     end if
24: end if
25: return $state$
```

3.2.2 Costs and benefits of merging.

As previously mentioned, the primary reason for state merging is to reduce duplicate work. The best candidates for merging have very similar path constraints. When path constraints vary greatly, the resulting symbolic expression places stresses upon the SMT solver. The symbolic expressions are not only larger, but also contain disjunctions. SMT solvers perform poorly on symbolic expressions containing disjunctions.

To assess the effects of merging, this research enforces a limit on the number of states that can merge into a single state. The two levels tested in this work are 5 states and 20 states. It is expected that at a certain point the symbolic expressions will become too complex to experience a benefit from merging.

3.3 State Pruning

State pruning attempts to reduce the amount of duplicate work performed during exploration. If two states arrive at the same program point and are equivalent, only one state must be explored further. Depending on the objective of the specific task, varying levels of strictness can be applied to determine the equivalence of two states. By relaxing equivalence requirements, the tool may under-approximate the behavior of the system. That is, two states that would exercise different areas of code may be deemed to be equivalent; thus, a portion of code would go unexplored. This may be acceptable in the case where exhaustive exploration is infeasible. While the under-approximation would prevent a specific area of code to go unexplored, the pruning may allow the symbolic execution tool to reach other areas of code that would have been otherwise impossible. This research explores this tradeoff in Chapter 4.

For this research, two states are equivalent if

1. the path constraints for each state are the same,

2. the call stacks for each state are the same, and

3. memory is the same.

The motivation for these criterion is that if two states have the same symbolic constraints, the two states will exercise the same portion of code. Therefore, only one state must be explored further.

The way this research implements state pruning is by maintaining a history of each state visited during execution. To reduce space requirements, a snapshot of the state is taken before adding it to the history. Snapshots only include the information relevant for determining if two states are equivalent is stored.

The state pruning technique is referred to as KLEE-PRUNE in the remainder of this work.

Algorithm 3.2 provides state pruning in algorithmic notation.

Algorithm 3.2 State Pruning - selectState()

```
Require: baseSearcher
 1: // snapshotHistory - snapshot of all states visited
 2:
 3: $state \leftarrow$ baseSearcher.selectState()
 4: $stateSnapshot \leftarrow$ createSnapshot($state$)
 5: if $snapshotHistory$.contains($stateSnapshot$) then
 6:    prune($state$)
 7:    return selectState()
 8: else
 9:    $snapshotHistory$.add($stateSnapshot$)
10:    return $state$
11: end if
```

3.4 Combining state merging and state pruning

The techniques of state merging and state pruning are not mutually exclusive. That is, the two different strategies can be combined with the hope of additional reduction to the state space. When two states arrive at the same program point, the algorithm first checks to see if the states are equivalent. If the states are equivalent, one of the states can be pruned.

If the states are not equivalent, the algorithm then checks to see if the two states can be merged. Note, pruning occurs first because it leads to the greatest reduction. In addition, once states have merged it is very likely that they will be pruned in the future because of the added complexity of the path constraint.

Based on results from pilot testing, state merging with pruning is only tested with the maximum number of merges set at 5 for a single state and fast-forward functionality enabled.

The combined techniques of state merging and state pruning is referred to as KLEE-PRUNE-MERGE in the remainder of this work. Algorithm 3.3 provides state merging with pruning in algorithmic notation.

3.5 System Boundaries

The SUT is the KLEE symbolic execution tool shown in Figure 3.2. The workload submitted to the system is the GNU COREUTILS utility suite and symbolic input for the utility to process. The system performs the analysis and returns statistics relating to the search and per path test cases to reproduce the given execution path.

3.5.1 KLEE Flags.

The flags used were chosen in an attempt to mimic the results of the original KLEE paper [7]. Due to changes in the KLEE tool, the following commands are the closest to the original test [30].

```
klee --output-dir <output_dir> --simplify-sym-indices \
--write-cvcs --write-cov --output-module \
--max-memory=4096 --disable-inlining --optimize \
--use-forked-solver --use-cex-cache --libc=uclibc \
--posix-runtime --allow-external-sym-calls \
--only-output-states-covering-new --environ=test.env \
--run-in=/tmp/sandbox --max-sym-array-size=4096 \
--max-instruction-time=30. --max-time=3600 --watchdog \
--max-memory-inhibit=false --max-static-fork-pct=1 \
--max-static-solve-pct=1 --max-static-cpfork-pct=1 \
```

Algorithm 3.3 State Merge combinded with State Pruning - selectState()

```
Require: baseSearcher
 1: // fastForwardState - state selected to be fast-forwarded
 2: // snapshotHistory - snapshot of all states visited
 3:
 4: if $fastForwardCount > 0$ then
 5:     $state \leftarrow fastForwardState$
 6: else
 7:     $state \leftarrow$ baseSearcher.selectState()
 8: end if
 9: $stateSnapshot \leftarrow$ createSnapshot($state$)
10: if $snapshotHistory$.contains($stateSnapshot$) then
11:     prune($state$)
12:     return selectState()
13: end if
14: for all states $s$ at $state.inst$ do
15:     if canMerge($state$, $s$) then
16:         // Merges s into state
17:         doMerge(state, s)
18:         // Disable fast-forward mode
19:         $fastForwardMode \leftarrow 0$
20:         Delete $s$
21:         return $state$
22:     end if
23: end for
24: if $fastForwardCount \leq 0$ then
25:     // Returns true if $state$ is found in the history of another active state
26:     if canFastForward($state$) then
27:         $fastForwardCount \leftarrow 10$
28:     end if
29: end if
30: return $state$
```

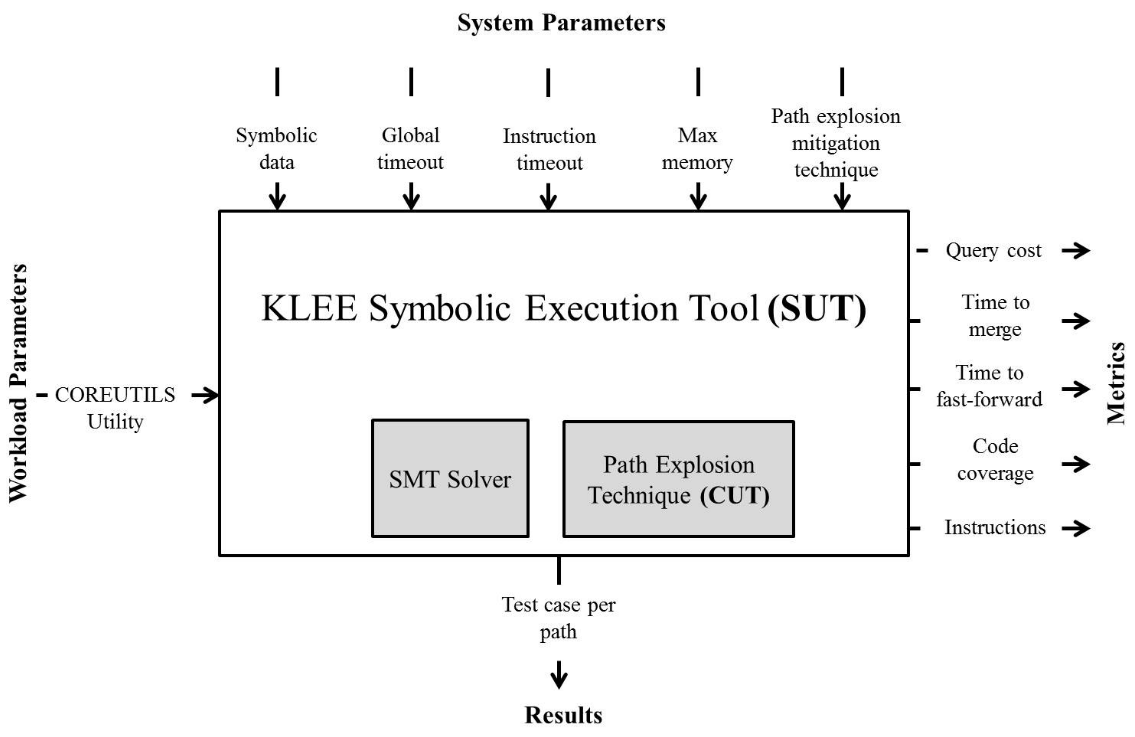


Figure 3.2: System Under Test (SUT).

```
--switch-type=internal --write-sym-paths  <searcher> \
./<utility>.bc --sym-args 0 1 10 --sym-args 0 2 2 \
--sym-files 1 8 --sym-stdout
```

3.5.2 Reducing non-determinism.

To allow for a more direct comparison of the path mitigation techniques with KLEE-BASE and reduce the number of experiment replications needed, this research attempts to remove as much non-determinism from the KLEE system as possible. Sources of non-determinism include:

- address space layout randomization (ASLR)
- `--randomize-fork` flag
- random path selection used by the KLEE Searcher

This research disables ASLR on the host operating system. This allows memory allocations to occur in a more deterministic fashion. Thus, replications of the experiments should experience less variance.

The `--randomize-fork` flag forces to the tool to randomly switch the true and false states at a fork [13]. This change removes an additional source of non-determinism and is not expected to drastically change the performance of KLEE.

The optimal configuration for KLEE-BASE is an interleaved search strategy that switches between a NURS algorithm favoring code coverage and a random path searcher. To increase the deterministic behavior of the system, KLEE-BASE is run without the random path searcher. This causes KLEE-BASE to perform below its maximum potential but allows for better comparison between the path mitigation techniques under test.

3.6 Workload

The workload submitted to the system for this research is the GNU COREUTILS 6.10 utility suite [3]. COREUTILS suite is chosen because it is used in the original KLEE research [7]. The utility under analysis directly affects the performance of the KLEE tool. Large applications are inherently more complex. That is, the number of possible paths and the length of the paths is larger, and so KLEE must do more work to explore the state space.

3.6.1 GNU COREUTILS.

The COREUTIL suite of tools represent real-world code that are used on a daily basis by Unix/Linux users. The utility suite includes a variety of different classes of utilities including file utilities, text utilities and shell utilities. Table 3.1 shows the exact utilities submitted to the SUT.

Table 3.1: List of GNU COREUTILS tested.

base64	dd	id	mv	rm	touch
chcon	df	join	nice	rmdir	tr
chgrp	dircolors	kill	nohup	seq	tsort
chmod	du	link	od	setuidgid	tty
chown	env	ln	paste	shred	uname
cksum	expand	logname	patchk	shuf	unexpand
comm	factor	ls	pinky	sleep	uniq
cp	fmt	mkdir	pr	split	unlink
csplit	fold	mkinfo	printf	stty	wc
cut	head	mknod	ptx	tail	who
date	hostid	mktemp	readlink	test	whoami

The COREUTILS suite includes a variety of applications ranging from 47 LOC to 3247 LOC. For the purposes of this research, a LOC is defined to be a single line that contains at least one program statement (comments and white space is excluded from the LOC count). Figure 3.3 provides a histogram for individual utilities in the suite. The total line count for the utilities tested is 33.4 thousands (kilo) of lines of code (KLOC).

3.7 Performance Metrics

The primary objective of this research is to assess the effectiveness of state merging and state pruning in regards to increasing KLEE's ability to address the path explosion problem.

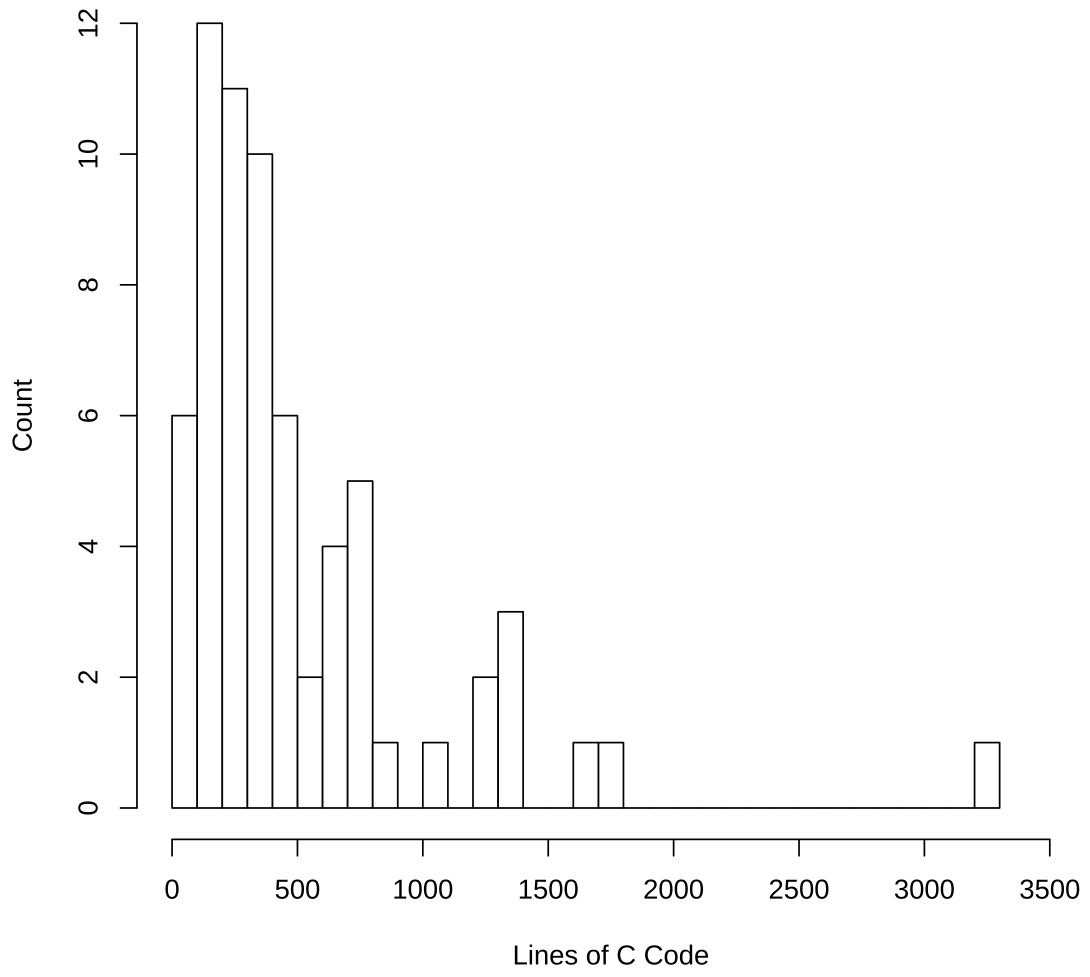


Figure 3.3: Histogram of LOC for COREUTILS [3].

The system is evaluated based on the following performance metrics: code coverage, instruction reduction, path reduction, average query cost, merge time, and fast forward time.

3.7.1 Code coverage.

The methodology this research uses to compare code coverage differs from the previous KLEE work. The KLEE tool writes out a test case containing concrete values for each path that will reproduce the given path. However, when merging states, KLEE will only write out a test case that guides execution down one of the many potential paths that are merged together. Since this research is concerned with a comparison of KLEE-BASE with the path mitigation techniques and is not interested in the maximum possible code coverage of a specific application, a different approach is taken.

This work uses the LLVM bitcode instructions for measuring the performance of the different algorithms. Code coverage, as a percentage, is the unique number of LLVM bitcode instructions executed divided by the total number of LLVM bitcode instructions in the compiled utility.

When concerned with absolute code coverage, this method is problematic because the compiled LLVM bitcode includes any libraries required by the application. Thus, code from the external libraries that will never be called is included in the overall size of the code. This will result is lower than expected code coverage percentages. However, since this research is concerned with a relative comparison between algorithms and not an absolute value for code coverage, this is deemed acceptable.

3.7.2 Instruction reduction.

While overall an increase in code coverage is the main objective of the path reduction techniques, instruction reduction provides insight into the amount of savings gained by merging and pruning states. KLEE maintains the total number of instructions executed during a run. Note, this count is not unique instructions. Therefore, a reduction in the number of instructions executed with an increase in code coverage demonstrates a reduction in duplicate instructions during execution.

3.7.3 Path reduction.

As with instruction reduction, path reduction provides insight into the amount of savings gained by merging and pruning states. A reduction in the number of paths executed with an increase in code coverage demonstrates a reduction in duplicate paths explored during execution.

3.7.4 Average query cost.

The average query cost is the total number of constraints that are added during execution divided by the total number of queries sent to the SMT solver. This is a measure of the additional stress placed on the solver by state merging.

3.7.5 Merge time.

Determining if two states are suitable for merging is an additional overhead not experienced by KLEE-BASE. The merge time is the time spent checking if two states can be merged and the subsequent time to merge the states. Successful merges reduce the number of paths the symbolic execution tool must track. In this way, the overhead of successful merges can be overcome by the reduction in paths. All failed merges are pure overhead. That is, the time spent processing a failed merge is completely wasted time. For an algorithm to be effective, the merge success rate and especially the merge failure time must be low with respect to the total run time.

3.7.6 Fast forward time.

Two states must be at the same program point to attempt a merge. The fast-forward functionality searches for states that have executed the same instruction in the near past. This is an additional overhead not experienced by KLEE-BASE. Unlike a failed merge, a failed fast-forward is not pure overhead. Giving priority to a single state for a period of time will override the underlying search heuristic, such as code coverage. However, execution continues to move forward even with a failed fast-forward attempt.

3.8 System Parameters

The system parameters for this study are as follows:

- **Size of symbolic data**

  KLEE must be provided with some amount of symbolic data to begin exploring the state space. The original KLEE work uses the following command for symbolic input into the majority of the COREUTILs: `--sym-args 0 1 10 --sym-args 0 2 2 --sym-files 1 8 --sym-stdout`. This represents one long option, two small options, and two small input streams (stdin and one file). This amount of symbolic data is expected to be sufficient to achieve high code coverage [30].

  The original KLEE research determined that the above symbolic input does not produce satisfactory coverage for the COREUTILs listed below [7, 30]. The modified symbolic input is taken from the original KLEE work and is listed below.

```
dd          : --sym-args 0 3 10 --sym-files 1 8
              --sym-stdout

dircolors   : --sym-args 0 3 10 --sym-files 2 12
              --sym-stdout

mknod       : --sym-args 0 1 10 --sym-args 0 3 2
              --sym-files 1 8 --sym-stdout

od          : --sym-args 0 3 10 --sym-files 2 12
              --sym-stdout

pathchk     : --sym-args 0 1 2 --sym-args 0 1 300
              --sym-files 1 8 --sym-stdout

printf      : --sym-args 0 3 10 --sym-files 2 12
              --sym-stdout
```

- **Global timeout**

  KLEE provides a global timeout mechanism for the testing of a given utility. A timeout is required for cases where KLEE is unable to perform a complete exploration of the state space. This research follows the original KLEE research [7] and fixes the global timeout at one hour. Note, KLEE is able to complete the testing of some applications under the one hour time limit.

- **Instruction timeout**

  KLEE provides a timeout for each instruction that the tool handles. Most often, the majority of this time will be spent in the SMT solver. To prevent a single instruction from dominating the exploration, this research follows the original KLEE research and fixes the per instruction timeout at 30 seconds.

- **Maximum memory**

  Symbolic execution is computationally expensive from a memory requirement perspective. KLEE refuses to fork when the maximum memory capacity is met. For this research, the maximum memory is set at 4GB.

- **Path mitigation technique**

  The path mitigation technique is the plugin specifically designed to address the path explosion problem. Changing the technique affects overhead within the system, as well as potential improvements in performance by reducing duplicate work.

3.9 Factors

This research identifies three factors to measure the performance of each path mitigation technique. Table 3.2 summarizes the factors and their levels.

- **Path mitigation technique**

  Four path mitigation techniques are used: KLEE-BASE (no mitigation), state merging, state pruning and state merging with state pruning. The path mitigation techniques require additional overhead not faced by the baseline configuration and are weighed against any benefit gained from the individual technique.

- **Max merges**

  For the tests using state merging, the algorithm uses two different levels for the maximum number of merges per state: 5 states and 20 states. These levels are chosen to approximate an optimal balance between reduction in the state space and complexity introduced by merging states.

- **Fast-forward**

  A fast-forward strategy attempts to increase the chances of arriving at a potential merge point. Each execution state maintains a history of its past program points. Each newly selected state searches the history of all active execution states to look for a match. If a match is found, the state is put into fast-forward mode. To test the effectiveness of fast-forward strategy, this research tests the state merging algorithm with and without the fast-forwarding strategy.

Table 3.2: Experimental Factors.

	Level I	Level II	Level III		Level IV
Path mitigation	KLEE-BASE	State pruning	State merging		State merging with state pruning
Max merges per state	-	-	5	20	5
Fast-forward	-	-	yes	no	yes

3.10 Evaluation Technique

The evaluation technique used for this study is direct measurement of the KLEE symbolic execution tool. Direct measurement of the system is used since this research focuses on the performance of KLEE on real-world applications. The test equipment for the experiment consists of a Windows 7 host computer, VMWare 10.1. and Ubuntu 12.04.1 LTS Guest.

3.11 Experimental Design

This research employs a full factorial design. Given the factors levels, 7 unique experiments are needed. Based on pilot studies, the variation between experiments is expected to be low. For that reason, 3 repetitions of each experiment is conducted. This yields 3 repetitions $\times$ 7 unique experiments = 21 experiments. Since 66 unique COREUTILS are submitted to the system and run for one hour each, testing lasts 21 $\times$ 66 = 1386 hours.

3.12 Methodology Summary

This research proposes a method to mitigate the current path explosion problem faced by the symbolic execution tools. The System Under Test (SUT) is the KLEE program analysis tool, and the Component Under Test (CUT) is the path explosion mitigation technique. The techniques tested are state merging and state pruning. The research approach is to submit symbolic input to various COREUTILS utilities. The effectiveness of each path mitigation technique is measured by code coverage, SMT solver time, average query cost, merge time, and fast-forward time.

IV. Results and Analysis

This research introduces a novel algorithm to address the path explosion problem of symbolic execution. The novel algorithm combines the previously researched state reduction techniques of state merging [20] and state pruning [21]. This research implements a prototype of each algorithm within KLEE [10]. The algorithm prototypes run on 66 GNU COREUTILS [3] for one hour.

This chapter reports the results of the path mitigation techniques and provides an analysis of the techniques with respect to KLEE-BASE. The different experiments are compared based on overall code coverage of LLVM instructions, solver time, average query cost and overhead time introduced by merging and pruning.

4.1 Overview

This research tests four algorithms: unmodified KLEE (KLEE-BASE), state merging (KLEE-MERGE), state pruning (KLEE-PRUNE), and state merging combined with state pruning (KLEE-PRUNE-MERGE). The state merging algorithms require two additional parameters: maximum number of merges allowed for a single state and fast-forward functionality. The levels chosen for the maximum number of merges are 5 and 20. Fast-forward functionality attempts to increase the probability that two states arrive at the same program point. KLEE-MERGE-5 refers to the state merging algorithm with a maximum of 5 merges into a single state and fast-forward functionality enabled. KLEE-MERGE-20 refers to the state merging algorithm with a maximum of 20 merges into a single state and fast-forward functionality enabled. KLEE-NO-FF-MERGE-5 refers to the state merging algorithm with a maximum of 5 merges into a single state and fast-forward functionality disabled. KLEE-NO-FF-MERGE-20 refers to the state merging algorithm with a maximum of 20 merges into a single state and fast-forward functionality disabled. Table 4.1 provides a summary of the coverage results for each algorithm. Note, the code coverage and instructions results are in terms of the compiled LLVM bitcode instructions.

Appendix A provides the code coverage for each COREUTIL and algorithm pair compared to KLEE-BASE. In general, the algorithms perform poorly with respect to code coverage on the same COREUTILs. The utilities that the algorithms perform poorly on are from each class of utilities: file system, process, text and shell.

Table 4.1: Summary of Coverage Results for All Algorithms.

Statistic	Average Cov. (%)	Lowest Cov. (%)	Highest Cov. (%)
KLEE-BASE	19.27	6.98	33.7
KLEE-MERGE-5	20.42	4.91	33.7
KLEE-MERGE-20	19.55	4.89	33.6
KLEE-NO-FF-MERGE-5	19.32	6.77	34.4
KLEE-NO-FF-MERGE-20	18.65	6.61	33.1
KLEE-PRUNE	20.69	6.63	33.6
KLEE-PRUNE-MERGE	19.43	5.06	29.9

4.2 KLEE-BASE

This research uses KLEE-BASE, an unmodified version of KLEE, as the baseline measurement for each COREUTIL. KLEE-BASE uses the coverage oriented search heuristic (`-search=nurs:covnew`).

4.3 Merging Results

The four state merging algorithms outperform KLEE-BASE on 51 of the 66 COREUTILS. KLEE-MERGE-5 is the only merging algorithm to average better code coverage than KLEE-BASE. KLEE-NO-FF-MERGE-20 performs worse than all of the other merging algorithm and also performs worse than KLEE-BASE. This supports the idea that over merging states can create too much complexity for the SMT solver to achieve a performance increase. Appendix A shows the merging algorithms do not excel at a particular class of utilities with respect to code coverage. Instead, the improvements over KLEE-BASE come from each class of utilities.

4.3.1 KLEE-MERGE-5.

KLEE-MERGE-5 is the state merging algorithm with a maximum of five merges per state and fast-forward functionality enabled. This algorithm performs the best of the state merging algorithms with respect to code coverage.

Table 4.2 provides a summary of the results. Note, negative values in the reduction column corresponds to an increase rather than a reduction in the particular statistic.

Table 4.2: Impact of KLEE-MERGE-5 on Average Code Coverage, Instructions, Queries, Query Constructs, Average Query Cost, Successful Merges, and Successful Fast-Forward.

Statistic	KLEE-BASE	KLEE-MERGE-5	Reduction (%)
Average Coverage (%)	19.27	20.42	-
Instructions	63,214,630	57,082,827	9.7
Queries	920,246	580,014	37.0
Query Constructs	486,143,051	885,297,982	-82.1
Average Query Cost	528.28	1,526.34	-189
SMT Solver Time (%)	97.8	97.0	0.82
Successful Merges	-	86.9	-
Merge Time (%)	-	1.44	-
Successful Fast-Forward	-	67.9	-
Fast-Forward Time (%)	-	0.25	-

4.3.1.1 Code Coverage.

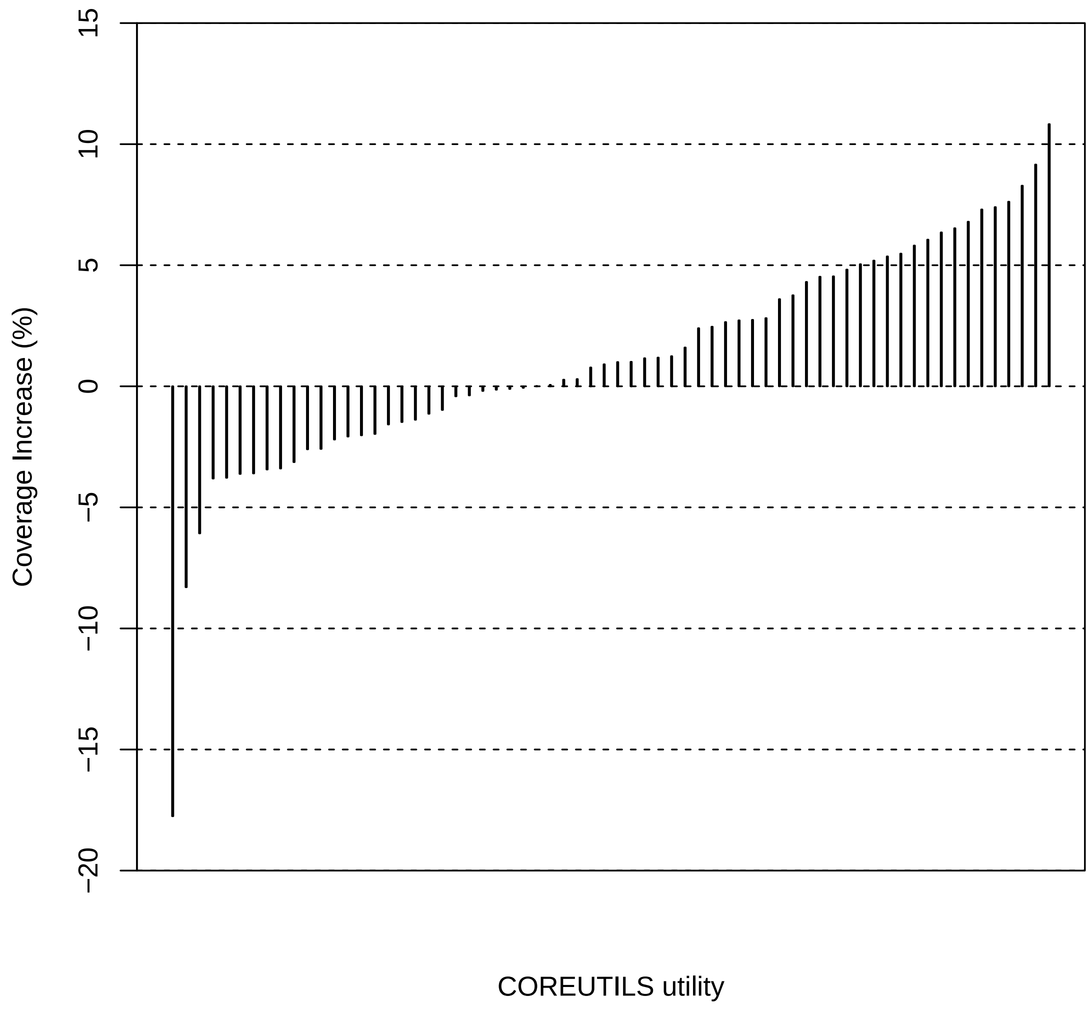


Figure 4.1: Coverage Increases over COREUTILS for KLEE-MERGE-5.

KLEE-MERGE-5 outperforms KLEE-BASE on 39 of the 66 COREUTILS (~59%). The average coverage increase for KLEE-MERGE-5 is 20.42% of the LLVM bitcode instructions. Figure 4.1 shows the coverage increase for each COREUTIL.

4.3.1.2 Instruction Reduction.

The average over the runs shows a 9.7% reduction in the number of total instructions executed. The merging of two states reduces the amount of duplicate instructions executed. Two separate states that execute over the same portion of code will require each instruction to be handled twice. By merging the two states together when possible, the instructions must only be handled once.

4.3.1.3 Average Query Cost.

The number of queries sent to the SMT solver decreased by 37%. However, the average query cost for KLEE-MERGE-5 was higher than KLEE-BASE. Therefore, while the number of queries sent to the SMT solver decreased, the queries sent by KLEE-MERGE-5 are larger and more complex. This result supports the notion that state merging creates more complex states. As shown above, this complexity is overcome in ~59% of the COREUTILS.

4.3.1.4 SMT Solver Time.

KLEE-MERGE-5 spent 97.0% of the execution time in the SMT solver, which is 0.8% less time than KLEE-BASE. Note, due to execution that finished very quickly and led to outlier values, 97.0% is obtained by taking the average of the median values for each run. This result is somewhat unexpected due to the complexity introduced by merging states. However, the reduction in overall queries sent to the SMT solver accounts for the decrease in time spent inside the solver.

4.3.1.5 Merge and Fast-Forward Success.

KLEE-MERGE-5 averaged 86.9 merges per utility and 67.9 successful fast-forward attempts. The time to check if two states are eligible to merge and to perform the merge on states that are eligible accounts for 1.44% of the total execution time. The time spent searching for fast-forward candidates accounts for even less time, 0.25% of the total execution time. This indicates that the algorithm itself does not introduce too much overhead looking for opportunities and trying to merge. Note, this overhead is different from the overhead introduced to the SMT solver by more complex states.

4.3.2 KLEE-MERGE-20.

KLEE-MERGE-20 is the state merging algorithm with a maximum of twenty merges per state and fast-forward functionality enabled. Table 4.3 provides a summary of the results.

Table 4.3: Impact of KLEE-MERGE-20 on Average Code Coverage, Instructions, Queries, Query Constructs, Average Query Cost, Successful Merges, and Successful Fast-Forward.

Statistic	KLEE-BASE	KLEE-MERGE-20	Reduction (%)
Average Coverage (%)	19.27	19.55	-
Instructions	63,214,630	17,026,253	73.1
Queries	920,246	281,005	69.5
Query Constructs	486,143,051	906,872,707	-86.5
Average Query Cost	528.28	3,227.25	-511
SMT Solver Time (%)	97.8	98.0	-0.2
Successful Merges	-	286	-
Merge Time (%)	-	2.5	-
Successful Fast-Forward	-	219.73	-
Fast-Forward Time (%)	-	0.11	-

4.3.2.1 Code Coverage.

KLEE-MERGE-20 outperforms KLEE-BASE on 35 of the 66 COREUTILS. The average code coverage for KLEE-MERGE-20 is 19.55% of the LLVM bitcode instructions,

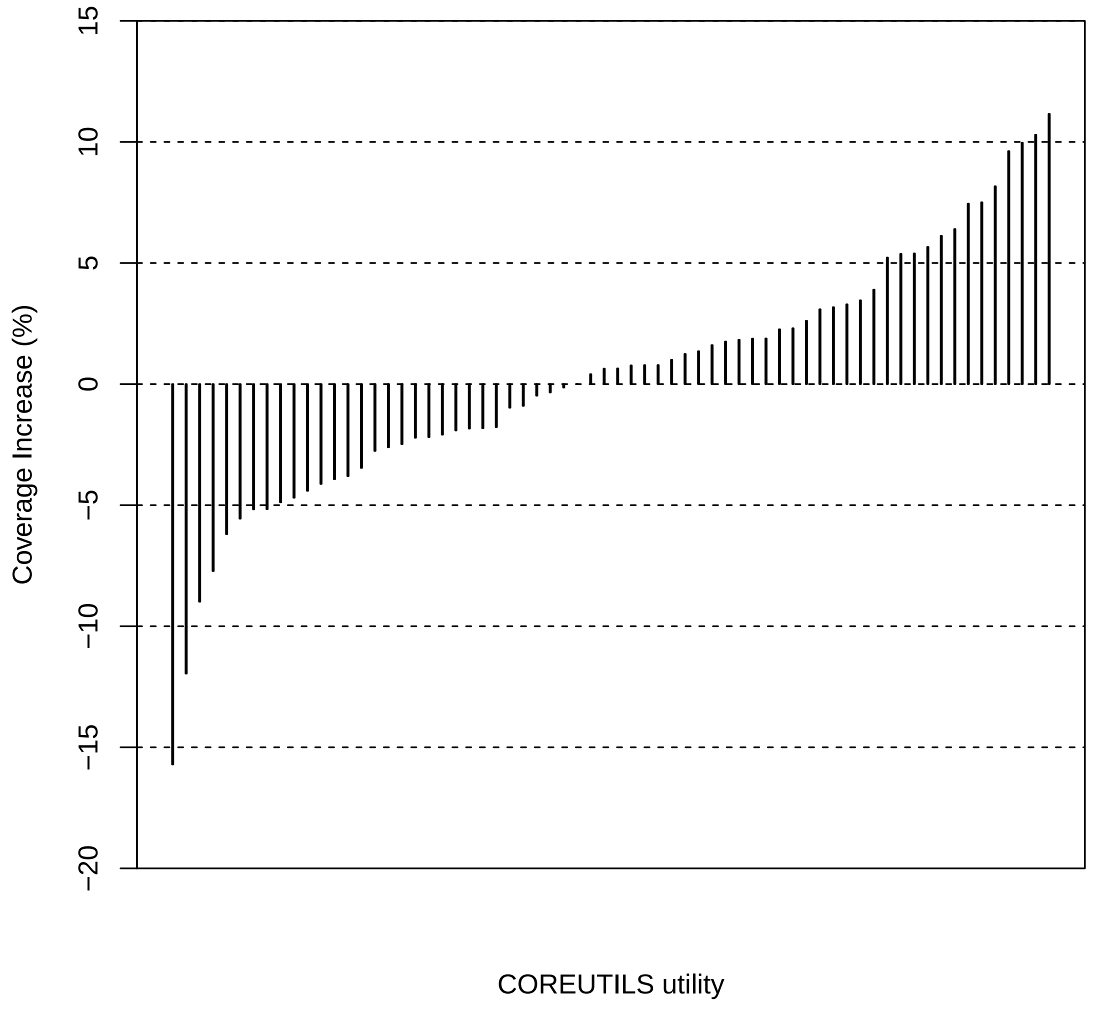


Figure 4.2: Coverage Increases over COREUTILS for KLEE-MERGE-20.

marginally higher than KLEE-BASE. Figure 4.2 shows the coverage increase for each COREUTIL.

4.3.2.2 Instruction Reduction.

The average over the runs shows a 73.1% reduction in the number of instructions executed. The reduction in instructions by KLEE-MERGE-20 is significantly higher than that of KLEE-MERGE-5. This is supported by the increase in successful merges with KLEE-MERGE-20 at 286 and KLEE-MERGE-5 at 86.9. That is, the increased number of merges reduces the number of executions of duplicate instructions.

4.3.2.3 Average Query Cost.

The number of queries sent to the SMT solver decreased by 69.5%. However, the average query cost for KLEE-MERGE-20 was higher than KLEE-BASE. Therefore, while the number of queries sent to the SMT solver decreased, the queries sent by KLEE-MERGE-20 are larger and more complex. This result supports the notion that state merging creates more complex states. As shown above, this complexity is overcome in 53% of the COREUTILS.

4.3.2.4 SMT Solver Time.

KLEE-MERGE-20 spent 98.0% of the execution time in the SMT solver, which is 0.2% more time than KLEE-BASE. Since the tests were run for an hour, this difference amounts to approximately 0.12 seconds. This result differs from KLEE-MERGE-5, where the percentage of time spent in the SMT solver was lower than that of KLEE-BASE.

4.3.2.5 Merge and Fast-Forward Success.

KLEE-MERGE-20 averaged 286 merges per utility and 219.73 successful fast-forward attempts. The time to check if two states are eligible to merge and to perform the merge on states that are eligible accounts for 2.5% of the total execution time. The time spent searching for fast-forward candidates accounts for even less time, 0.11% of the total execution time. The increase in successful merges is due to the increase in the maximum number of states allowed to merge into one from 5 to 20. Therefore, it is likely KLEE-

BASE-5 tried to merge but failed because the maximum limit had been reached and not because the two states were incompatible.

4.3.3 KLEE-NO-FF-MERGE-5.

KLEE-NO-FF-MERGE-5 is the state merging algorithm with a maximum of five merges per state and fast-forward functionality disabled. Table 4.4 provides a summary of the results.

Table 4.4: Impact of KLEE-NO-FF-MERGE-5 on Average Code Coverage, Instructions, Queries, Query Constructs, Average Query Cost, SMT Solver Time and Successful Merges.

Statistic	KLEE-BASE	KLEE-NO-FF-MERGE-5	Reduction (%)
Average Coverage (%)	19.27	19.32	-
Instructions	63,214,630	20,127,369	68.2
Queries	920,246	649,544	29.4
Query Constructs	486,143,051	1,133,119,093	-133.1
Average Query Cost	528.28	1,744.48	-230.2
SMT Solver Time (%)	97.8	98	-0.2
Successful Merges	-	170.3	-
Merge Time (%)	-	1.2	-

4.3.3.1 Code Coverage.

KLEE-NO-FF-MERGE-5 outperforms KLEE-BASE on 38 of the 66 COREUTILS. The average KLEE-NO-FF-MERGE-5 is 19.32% of the LLVM bitcode instructions, slightly lower than KLEE-BASE. Figure 4.3 shows the coverage increase for each COREUTIL.

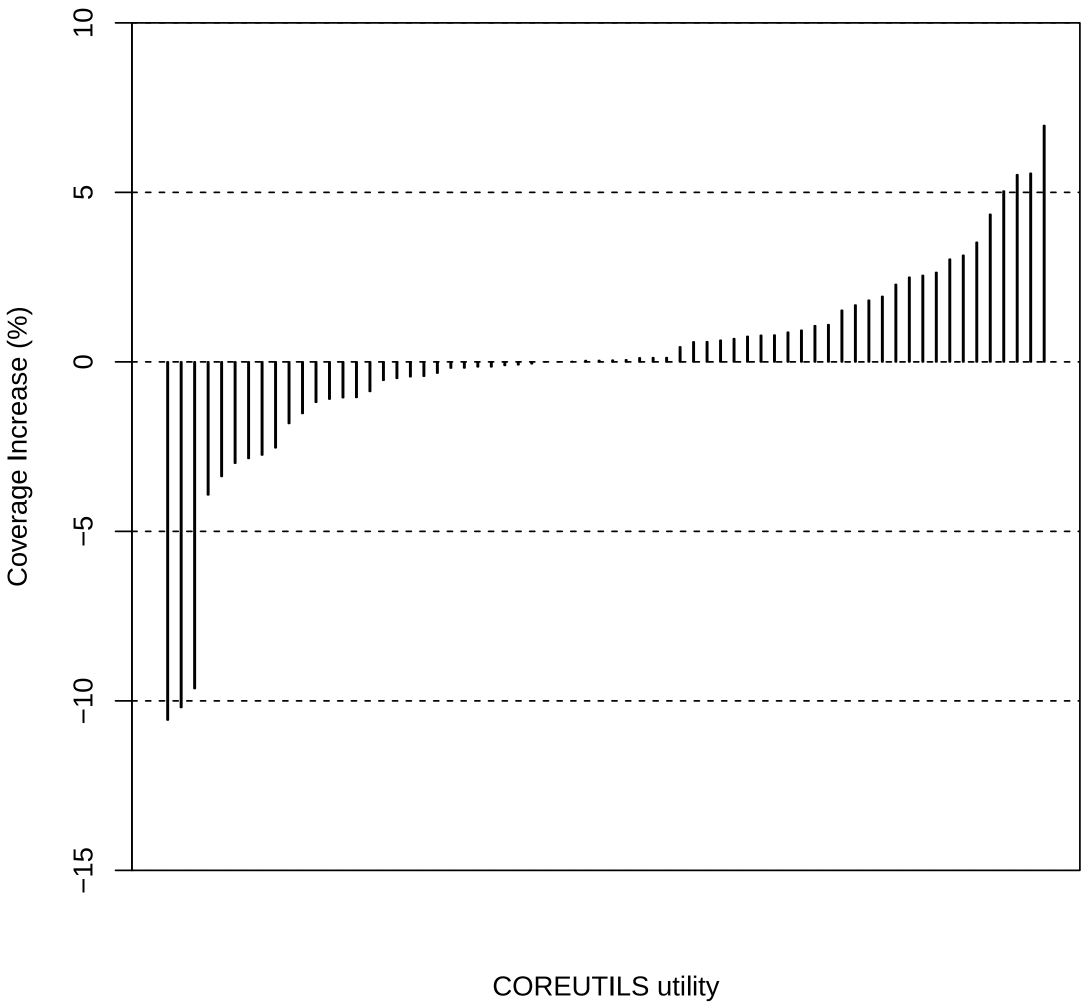


Figure 4.3: Coverage Increases over COREUTILS for KLEE-NO-FF-MERGE-5.

4.3.3.2 Instruction Reduction.

The average over the runs shows a 68.2% reduction in the number of instructions executed. These results are slightly higher than KLEE-MERGE-5, which is unexpected since KLEE-MERGE-5 would most likely have more successful merges with the addition of the fast-forward functionality. However, that is not the case. KLEE-NO-FF-MERGE-5 averaged 170.3 merges per utility while KLEE-MERGE-5 averaged 86.9 merges per utility.

4.3.3.3 Average Query Cost.

The number of queries sent to the SMT solver decreased by 29.4%. However, the average query cost for KLEE-MERGE-5 was higher than KLEE-BASE. These results are very similar to KLEE-BASE-5.

4.3.3.4 SMT Solver Time.

KLEE-NO-FF-MERGE-5 spent 98.0% of the execution time in the SMT solver, which is 0.2% more time than KLEE-BASE.

4.3.3.5 Merge Success.

KLEE-NO-FF-MERGE-5 averaged 170.3 merges per utility. This is higher than KLEE-MERGE-5, which averaged 86.9 merges per utility. This is unexpected due to the fact that KLEE-MERGE-5 uses fast-forward functionality that is designed specifically to increase the chances of a merge occurring. This research is unable to explain this discrepancy, although it may be due to the non-determinism of the system.

4.3.4 KLEE-NO-FF-MERGE-20.

KLEE-NO-FF-MERGE-20 is the state merging algorithm with a maximum of twenty merges per state and fast-forward functionality disabled. Table 4.5 provides a summary of the results.

4.3.4.1 Code Coverage.

KLEE-NO-FF-MERGE-20 outperforms KLEE-BASE on 24 of the 66 COREUTILS. The average code coverage for KLEE-NO-FF-MERGE-20 is 18.65% of the LLVM bitcode instructions. KLEE-NO-FF-MERGE-20 was the lowest performing merging algorithm with respect to code coverage. Figure 4.4 shows the coverage increase for each COREUTIL.

Table 4.5: Impact of KLEE-NO-FF-MERGE-20 on Average Code Coverage, Instructions, Queries, Query Constructs, Average Query Cost, SMT Solver Time and Successful Merges.

Statistic	KLEE-BASE	KLEE-NO-FF-MERGE-20	Reduction (%)
Average Coverage (%)	19.27	18.65	-
Instructions	63,214,630	6,028,078	90.5
Queries	920,246	267,285	71
Query Constructs	486,143,051	662,659,108	-36.3
Average Query Cost	528.28	2,479.22	-369.3
SMT Solver Time (%)	97.8	98.3	-0.5
Successful Merges	-	324.4	-
Merge Time (%)	-	0.77	-

4.3.4.2 Instruction Reduction.

The average over the runs shows a 90.5% reduction in the number of instructions executed. KLEE-NO-FF-MERGE-20 achieved the highest reduction of both instructions and paths out of all state merging techniques. However, as shown above, KLEE-NO-FF-MERGE-20 only increased overall code coverage on 24 of the 66 COREUTILS.

4.3.4.3 Average Query Cost.

The number of queries sent to the SMT solver decreased by 71%. , KLEE-NO-FF-MERGE-20 experienced the largest reduction in queries sent to SMT solver. KLEE-NO-FF-MERGE-20 also had the largest average query cost. So while the total number of queries sent decreased, the complexity of the queries that were sent was significantly higher than the other algorithms.

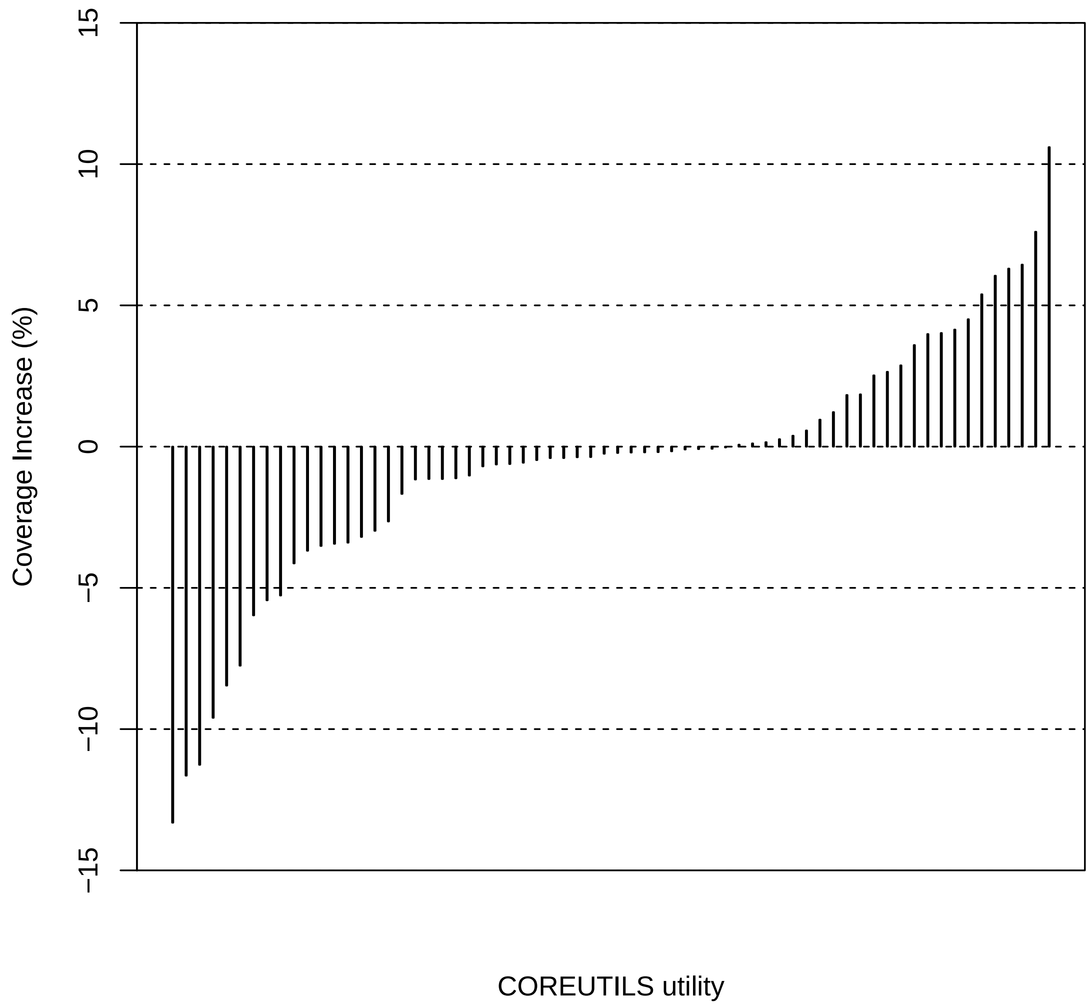


Figure 4.4: Coverage Increases over COREUTILS for KLEE-NO-FF-MERGE-20.

4.3.4.4 SMT Solver Time.

KLEE-NO-FF-MERGE-5 spent 98.3% of the execution time in the SMT solver, which is 0.5% more time than KLEE-BASE. This follows from the increase in average query cost, which represents more difficult queries of the SMT solver.

4.3.4.5 Merge Success.

KLEE-NO-FF-MERGE-20 averaged 324.4 merges per utility. Again, the merging strategy without fast-forwarding functionality was able to successfully merge more states; KLEE-MERGE-20 averaged 286 merges per utility.

4.3.5 Merging Summary.

At least one of the state merging techniques performed better than KLEE-BASE on 51 of the 66 COREUTILS. All four merging techniques were successful in reducing both the number of instructions executed and paths explored. However, the two algorithms that achieved the higher reduction in instructions and paths, KLEE-MERGE-20 and KLEE-NO-FF-MERGE-20, were also the worst performing. This is a result of the extra stress placed upon the SMT as shown by the average query cost of each algorithm. With a naïve merging technique that merges any eligible state without regard for the complexity of the resulting state, a maximum number of merges per state of 20 states is too large to experience an improvement over KLEE-BASE.

The fast-forward functionality proved to be unsuccessful in increasing the number of successful merges. The large number of fast-forward attempts had a low success rate and did not increase the number of successful merges. However, this research did not expect the difference between the number successful merges for the algorithms with and without the fast-forward functionality to be so large. This is most likely due to non-determinism within the system. Since this is the primary concern of this research, more testing is needed to verify this result.

4.4 Pruning Results

KLEE-PRUNE is an implementation of the state pruning algorithm. KLEE-PRUNE performed the best of the algorithms tested in this research. Appendix A shows that KLEE-PRUNE performed especially well on file system ownership manipulation (`chown`, `chmod`, `chgrp`). This does not translate to all file manipulation commands. For example, KLEE-PRUNE performed much worse than KLEE-BASE on `mv` and ls.

Table 4.6: Impact of KLEE-PRUNE on Average Code Coverage, Instructions, Queries, Query Constructs, Average Query Cost, SMT Solver Time and Pruned States.

Statistic	KLEE-BASE	KLEE-PRUNE	Reduction (%)
Average Coverage (%)	19.27	20.69	-
Instructions	63,214,630	32,375,332	48.8
Queries	920,246	810,350	11.9
Query Constructs	486,143,051	523,194,126	-7.6
Average Query Cost	528.28	645.64	-22.2
SMT Solver Time (%)	97.8	93.4	4.5
Pruned States	-	1,142	-
Prune Time(%)	-	4.03	-

4.4.1.1 Code Coverage.

As shown in Table 4.6, KLEE-PRUNE outperforms KLEE-BASE on 40 of the 66 COREUTILS. The average code coverage for KLEE-PRUNE is 20.69% of the LLVM bitcode instructions. KLEE-PRUNE performed the best of all algorithms tested with respect to code coverage. Figure 4.5 shows the coverage increase for each COREUTIL.

4.4.1.2 Instruction Reduction.

The average over the runs shows a 48.8% reduction in the number of instructions executed. This reduction in instructions is a result of pruned states. The pruned states reduce the number of duplicate instructions executed.

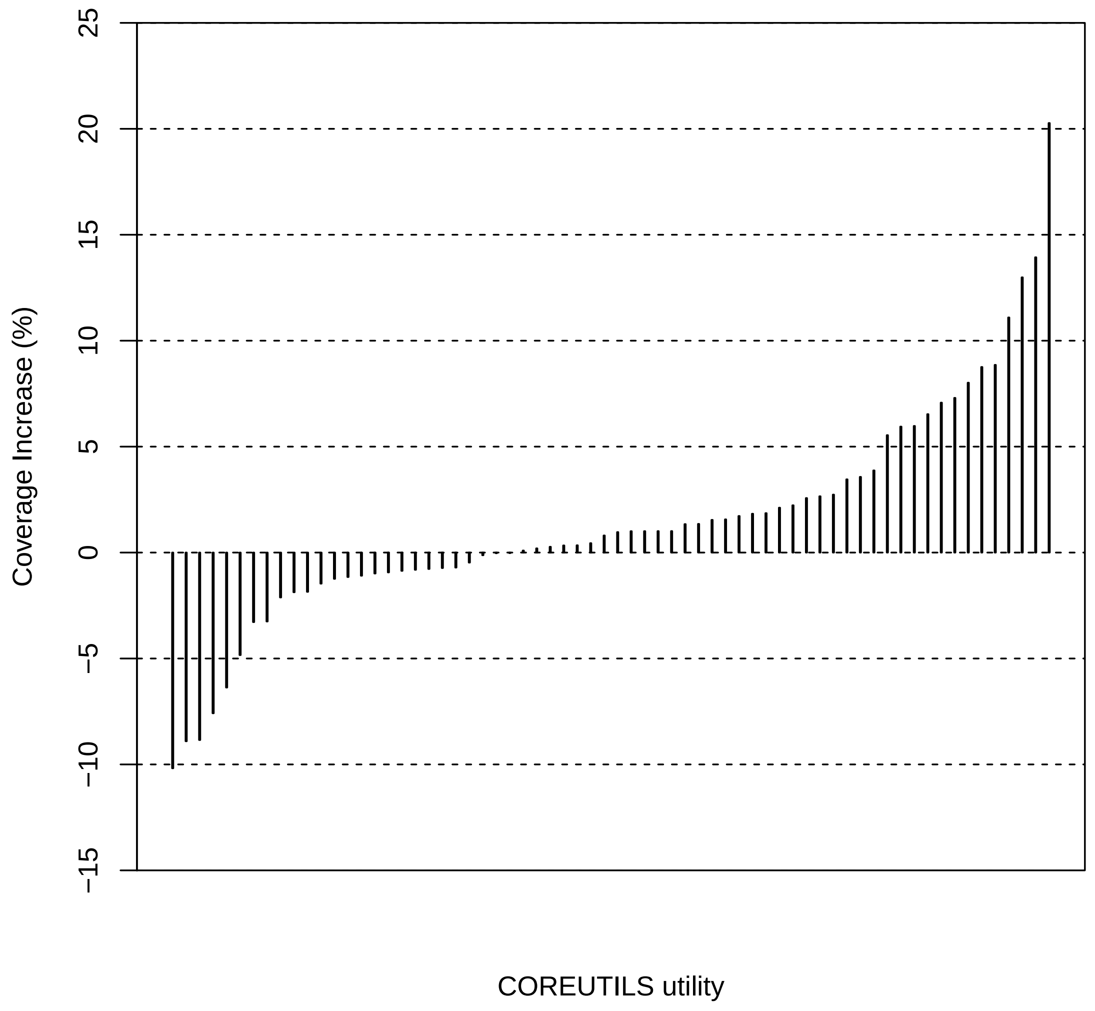


Figure 4.5: Coverage Increases over COREUTILS for KLEE-PRUNE.

4.4.1.3 Average Query Cost.

The number of queries sent to the SMT solver decreased by 11.9%. The average query cost for KLEE-PRUNE was higher than KLEE-BASE.

4.4.1.4 SMT Solver Time.

KLEE-PRUNE spent 93.4% of the execution time in the SMT solver, which is 4.4% less time than KLEE-BASE. This result is also lower than the merging techniques.

4.4.1.5 Pruned States.

KLEE-PRUNE prunes an average of 1,142 states per utility during exploration. Pruning accounts for 4.03% of the the execution time. This pruning is especially beneficial because unlike merging, pruning does not add additional strain to the SMT solver.

4.4.2 Pruning Summary.

KLEE-PRUNE performed the best with respect to code coverage of all the algorithms implemented. KLEE-PRUNE was able to successfully reduce the number of duplicate instructions executed by pruning redundant states, allowing for better code coverage on 40 of 66 COREUTILS. KLEE-PRUNE did especially well on the COREUTILS dealing with file system manipulation, as shown by Appendix A.

4.5 Pruning Combined with Merging Results

KLEE-PRUNE-MERGE is an implementation of the novel algorithm introduced in this research that combines state merging and pruning techniques. KLEE-PRUNE-MERGE did not perform as well as the individual KLEE-MERGE and KLEE-PRUNE algorithms with respect to code coverage. More work is needed to improve the synergy of the two separate techniques. KLEE-PRUNE-MERGE did perform well on COREUTILs from the class of text utilities (`expand`, `fmt`, `join`, `seq`). The remainder of this section presents the individual metrics for KLEE-PRUNE-MERGE.

4.5.1.1 Code Coverage.

As shown in Table 4.7, KLEE-PRUNE-MERGE outperforms KLEE-BASE on 36 of the 66 COREUTILS. The average code coverage for KLEE-PRUNE-MERGE is 19.46%

Table 4.7: Impact of KLEE-PRUNE-MERGE on Average Code Coverage, Instructions, Queries, Query Constructs, Average Query Cost, SMT Solver Time, Successful Merges, Successful Fast-Forward and Pruned States.

Statistic	KLEE-BASE	KLEE-PRUNE-MERGE	Reduction (%)
Average Coverage (%)	19.27	19.46	-
Instructions	63,214,630	40,938,388	35.2
Queries	920,246	641,973	30.2
Query Constructs	486,143,051	1,037,482,453	-113.4
Average Query Cost	528.28	1,616.08	-205.9
SMT Solver Time (%)	97.8	95.2	2.7
Successful Merges	-	84.2	-
Merge Time (%)	-	1.02	-
Successful Fast-Forward	-	67.1	-
Fast-Forward Time (%)	-	0.14	-
Pruned States	-	37.5	-
Prune Time (%)	-	0.17	-

of the LLVM bitcode instructions. Figure 4.6 shows the coverage increase for each COREUTIL.

4.5.1.2 Instruction Reduction.

The average over the runs shows a 35.2% reduction in the number of instructions executed. KLEE-PRUNE-MERGE does not achieve a high of an instruction reduction as KLEE-PRUNE. This suggests that merging states together into a single state decreases the probability of pruning that state in the future.

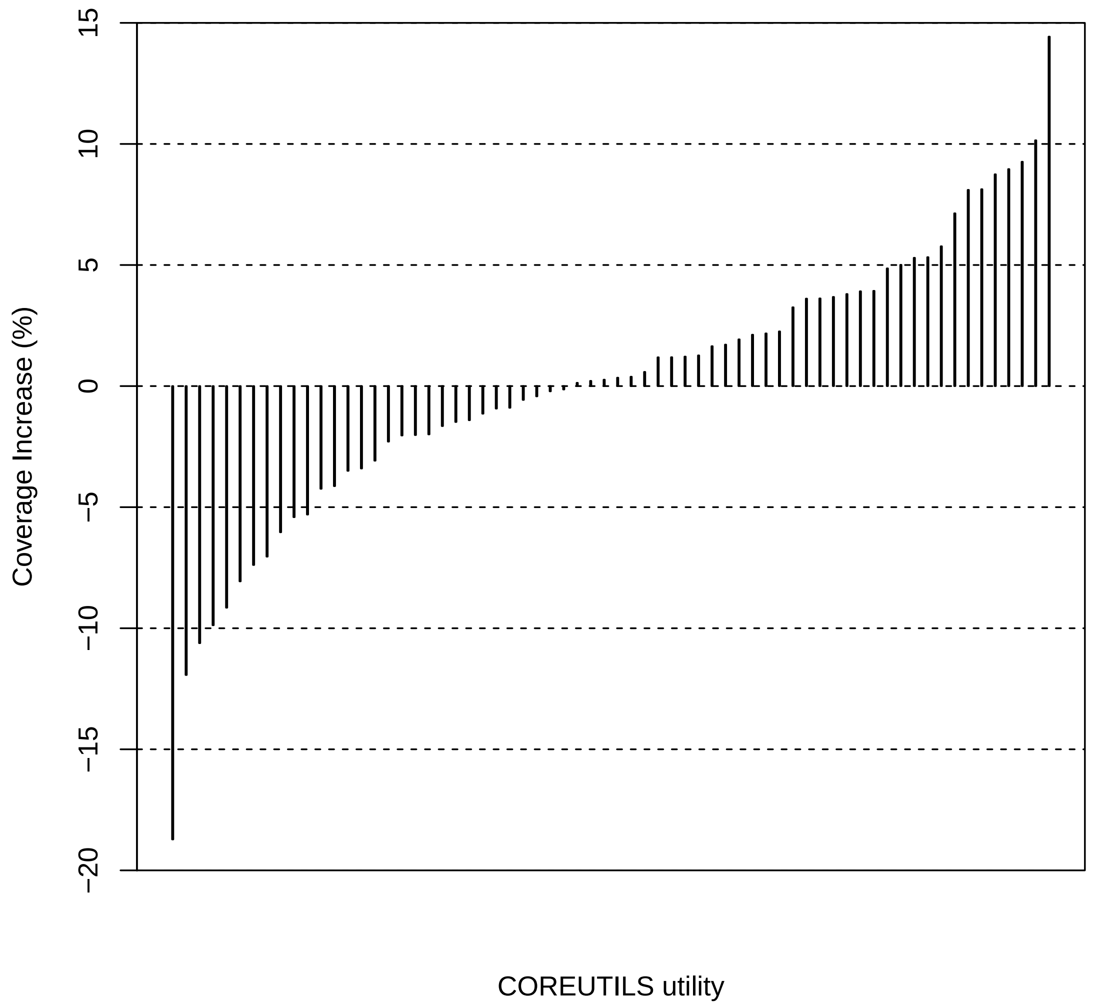


Figure 4.6: Coverage Increases over COREUTILS for KLEE-PRUNE-MERGE.

4.5.1.3 Average Query Cost.

The number of queries sent to the SMT solver decreased by 30.2%. However, the average query cost for KLEE-PRUNE-MERGE was higher than KLEE-BASE.

4.5.1.4 SMT Solver Time.

KLEE-PRUNE-MERGE spent 95.2% of the execution time in the SMT solver, which is 2.6% more time than KLEE-BASE.

4.5.1.5 Merge and Fast-Forward Success.

KLEE-PRUNE-MERGE averaged 84.2 merges per utility and 67.1 successful fast-forward attempts. The time to check if two states are eligible to merge and to perform the merge when possible accounts for 1.02% of the total execution time.

4.5.1.6 Pruned States.

KLEE-PRUNE-MERGE averaged 37.5 pruned states per utility. This reduction is more significant than merging because the pruned state is removed from the system completely, whereas, when two states are merged, information for both states is maintain in the system through the disjunction of the constraints, creating a more complex state. KLEE-PRUNE-MERGE was able to prune far fewer states than KLEE-PRUNE, 37.5 compared to 1142. This is likely due to the fact that once states are merged together, they are much less likely to be pruned in the future because of the added complexity of the path constraint.

4.5.2 Pruning Combined with Merging Summary.

KLEE-PRUNE-MERGE outperformed KLEE-BASE with respect to code coverage on 36 of the 66 COREUTILS. However, KLEE-PRUNE-MERGE was outperformed by the individual KLEE-MERGE and KLEE-PRUNE algorithms. KLEE-PRUNE-MERGE was unable to prune the same number of states as KLEE-PRUNE, negatively impacting the overall code coverage. Since KLEE-PRUNE-MERGE merges states together, this is a potential cause for the reduction in the number of pruned states.

4.6 Results Summary

This section presents the results of each state merging and state pruning algorithm. No single algorithm completely dominated over another. The results show that naïvely merging 20 states into a single state creates too much computational complexity for the

SMT solver to improve the code coverage over KLEE-BASE. The merging strategies where the maximum merge limit is 5 performed better than KLEE-BASE for more than 50% of the COREUTILS.

The results show that state merging combined with state pruning is a viable strategy to increase code coverage. However, KLEE-PRUNE-MERGE did not perform as well as the KLEE-PRUNE or KLEE-MERGE-5. This suggests more work is needed to increase the synergy between state merging and state pruning when combined.

V. Conclusions and Future Work

Software vulnerabilities place sensitive personal and financial information at risk. The need to produce quality software, free of bugs is compelling. Symbolic execution is a promising technique for improving the overall quality of software and reducing the occurrence of vulnerabilities. Current symbolic execution tools suffer from a path explosion problem where the possible number of paths grows exponentially with respect to loops and conditionals.

This research explored two state reduction techniques, state merging and state pruning, to address the path explosion problem of symbolic execution. This research also introduced a novel algorithm that combines dynamic state merging with dynamic state pruning.

5.1 Contributions

5.1.1 Primary Contribution.

This research proposes a novel algorithm to address the path explosion problem of symbolic execution. A prototype of the algorithm was implemented in the KLEE symbolic execution tool. Analysis of the prototype showed mix results. On 35 of the 66 COREUTILS, the prototype outperformed KLEE-BASE.

5.1.2 Secondary Contributions.

This research implemented prototypes of both state merging and state pruning techniques and provides analysis including changes to code coverage. No single algorithm completely dominated over another. It was clear from the results, that a maximum of 20 states merging together was too high for a naïve state merging algorithm that merged at every possible chance. The complexity introduced by merging this large number of states together is too much for the SMT solver to overcome.

5.2 Limitations

The state merging and state pruning algorithms were implemented within the KLEE symbolic execution tool. While the algorithms are general enough to apply other symbolic execution tools, the results of this work are highly dependent upon the KLEE architecture.

Additionally, each algorithm was compared against a sub-optimal configuration of KLEE. Namely, the random path searcher was excluded from the KLEE-BASE test. This choice was made in order to more directly compare the effects of state merge and state pruning.

5.3 Future Work

Future work includes testing the state merging and state pruning algorithm in additional symbolic execution tools to verify that the properties hold over different tools.

Secondly, the state merging and state pruning algorithms proposed are naïve algorithms used to demonstrate the possibility of combining state merging and state pruning. More sophisticated state merging techniques [20] exist that use additional knowledge of the system, including static passes prior to execution, to determine the estimated cost and benefit of merging. Thus, this technique can reduce the stress placed on the SMT solver. In addition, more sophisticated state pruning techniques [21] exist that are able to determine when two states are functionally equivalent and increase the amount pruning done during the search process.

Appendix A: Code Coverage (%) for all COREUTILS

UTILITY	ALGORITHM	COVERAGE (%) DIFFERENCE — ALGORITHM AND KLEE_BASE
base64	KLEE-MERGE-5	0.0119
	KLEE-MERGE-20	0.0612
	KLEE-NO-FF-MERGE-5	0.0353
	KLEE-NO-FF-MERGE-20	0.0185
	KLEE-PRUNE	0.0346
	KLEE-PRUNE-MERGE	0.0896
chcon	KLEE-MERGE-5	0.0078
	KLEE-MERGE-20	-0.0184
	KLEE-NO-FF-MERGE-5	-0.0054
	KLEE-NO-FF-MERGE-20	-0.0063
	KLEE-PRUNE	-0.0071
	KLEE-PRUNE-MERGE	-0.0202
chgrp	KLEE-MERGE-5	-0.0019
	KLEE-MERGE-20	-0.0517
	KLEE-NO-FF-MERGE-5	-0.0338
	KLEE-NO-FF-MERGE-20	-0.0369
	KLEE-PRUNE	0.0708
	KLEE-PRUNE-MERGE	-0.0704
chmod	KLEE-MERGE-5	-0.0362
	KLEE-MERGE-20	-0.0276
	KLEE-NO-FF-MERGE-5	0.0255
	KLEE-NO-FF-MERGE-20	-0.0048
	KLEE-PRUNE	0.0388
	KLEE-PRUNE-MERGE	-0.0604
chown	KLEE-MERGE-5	-0.0344
	KLEE-MERGE-20	-0.0178
	KLEE-NO-FF-MERGE-5	0.0004
	KLEE-NO-FF-MERGE-20	-0.0017
	KLEE-PRUNE	0.1394
	KLEE-PRUNE-MERGE	-0.0175
cksum	KLEE-MERGE-5	-0.0006
	KLEE-MERGE-20	-0.0014
	KLEE-NO-FF-MERGE-5	0.0000
	KLEE-NO-FF-MERGE-20	-0.1331
	KLEE-PRUNE	-0.0086
	KLEE-PRUNE-MERGE	-0.1643
comm	KLEE-MERGE-5	0.0762
	KLEE-MERGE-20	0.0997
	KLEE-NO-FF-MERGE-5	0.0064
	KLEE-NO-FF-MERGE-20	0.0630
	KLEE-PRUNE	0.0172
	KLEE-PRUNE-MERGE	0.0478
cp	KLEE-MERGE-5	0.0006
	KLEE-MERGE-20	0.0188
	KLEE-NO-FF-MERGE-5	-0.0010
	KLEE-NO-FF-MERGE-20	-0.0019
	KLEE-PRUNE	0.0101
	KLEE-PRUNE-MERGE	0.0614

UTILITY	ALGORITHM	COVERAGE (%) DIFFERENCE — ALGORITHM AND KLEE_BASE
csplit	KLEE-MERGE-5	-0.0158
	KLEE-MERGE-20	-0.0182
	KLEE-NO-FF-MERGE-5	-0.0018
	KLEE-NO-FF-MERGE-20	-0.0037
	KLEE-PRUNE	0.0134
	KLEE-PRUNE-MERGE	0.0385
cut	KLEE-MERGE-5	0.0100
	KLEE-MERGE-20	-0.0346
	KLEE-NO-FF-MERGE-5	-0.0285
	KLEE-NO-FF-MERGE-20	-0.0298
	KLEE-PRUNE	0.0876
	KLEE-PRUNE-MERGE	-0.0093
date	KLEE-MERGE-5	0.0431
	KLEE-MERGE-20	-0.0517
	KLEE-NO-FF-MERGE-5	-0.0182
	KLEE-NO-FF-MERGE-20	0.0414
	KLEE-PRUNE	-0.0099
	KLEE-PRUNE-MERGE	0.0013
dd	KLEE-MERGE-5	-0.0259
	KLEE-MERGE-20	0.0346
	KLEE-NO-FF-MERGE-5	0.0075
	KLEE-NO-FF-MERGE-20	0.0096
	KLEE-PRUNE	0.0081
	KLEE-PRUNE-MERGE	0.0021
df	KLEE-MERGE-5	0.0240
	KLEE-MERGE-20	-0.0380
	KLEE-NO-FF-MERGE-5	-0.0044
	KLEE-NO-FF-MERGE-20	0.0026
	KLEE-PRUNE	-0.0003
	KLEE-PRUNE-MERGE	0.0362
dircolors	KLEE-MERGE-5	0.0116
	KLEE-MERGE-20	0.0183
	KLEE-NO-FF-MERGE-5	0.0152
	KLEE-NO-FF-MERGE-20	-0.0062
	KLEE-PRUNE	0.0653
	KLEE-PRUNE-MERGE	0.0218
du	KLEE-MERGE-5	-0.1775
	KLEE-MERGE-20	-0.1571
	KLEE-NO-FF-MERGE-5	-0.0152
	KLEE-NO-FF-MERGE-20	-0.0846
	KLEE-PRUNE	-0.0637
	KLEE-PRUNE-MERGE	-0.1872
env	KLEE-MERGE-5	-0.0607
	KLEE-MERGE-20	-0.1195
	KLEE-NO-FF-MERGE-5	-0.1056
	KLEE-NO-FF-MERGE-20	-0.0960
	KLEE-PRUNE	0.0034
	KLEE-PRUNE-MERGE	-0.1062

UTILITY	ALGORITHM	COVERAGE (%) DIFFERENCE — ALGORITHM AND KLEE_BASE
expand	KLEE-MERGE-5	0.0027
	KLEE-MERGE-20	-0.0219
	KLEE-NO-FF-MERGE-5	0.0087
	KLEE-NO-FF-MERGE-20	0.0761
	KLEE-PRUNE	0.0223
	KLEE-PRUNE-MERGE	0.0875
factor	KLEE-MERGE-5	0.0482
	KLEE-MERGE-20	0.0064
	KLEE-NO-FF-MERGE-5	-0.0275
	KLEE-NO-FF-MERGE-20	-0.0265
	KLEE-PRUNE	-0.0094
	KLEE-PRUNE-MERGE	0.0165
fmt	KLEE-MERGE-5	0.0680
	KLEE-MERGE-20	0.0539
	KLEE-NO-FF-MERGE-5	0.0004
	KLEE-NO-FF-MERGE-20	0.0011
	KLEE-PRUNE	0.1110
	KLEE-PRUNE-MERGE	0.0393
fold	KLEE-MERGE-5	0.0265
	KLEE-MERGE-20	0.0817
	KLEE-NO-FF-MERGE-5	-0.0008
	KLEE-NO-FF-MERGE-20	-0.0008
	KLEE-PRUNE	0.0802
	KLEE-PRUNE-MERGE	0.0532
head	KLEE-MERGE-5	-0.0339
	KLEE-MERGE-20	0.1116
	KLEE-NO-FF-MERGE-5	0.0001
	KLEE-NO-FF-MERGE-20	0.0360
	KLEE-PRUNE	0.0554
	KLEE-PRUNE-MERGE	-0.0165
hostid	KLEE-MERGE-5	0.0274
	KLEE-MERGE-20	0.0640
	KLEE-NO-FF-MERGE-5	0.0504
	KLEE-NO-FF-MERGE-20	-0.0413
	KLEE-PRUNE	0.0886
	KLEE-PRUNE-MERGE	-0.0413
id	KLEE-MERGE-5	0.0740
	KLEE-MERGE-20	-0.0248
	KLEE-NO-FF-MERGE-5	0.0006
	KLEE-NO-FF-MERGE-20	-0.0167
	KLEE-PRUNE	0.0154
	KLEE-PRUNE-MERGE	-0.0057
join	KLEE-MERGE-5	-0.0197
	KLEE-MERGE-20	0.0000
	KLEE-NO-FF-MERGE-5	0.0078
	KLEE-NO-FF-MERGE-20	-0.0025
	KLEE-PRUNE	-0.0013
	KLEE-PRUNE-MERGE	0.0501

UTILITY	ALGORITHM	COVERAGE (%) DIFFERENCE — ALGORITHM AND KLEE_BASE
kill	KLEE-MERGE-5	0.0548
	KLEE-MERGE-20	0.0538
	KLEE-NO-FF-MERGE-5	0.0059
	KLEE-NO-FF-MERGE-20	0.0252
	KLEE-PRUNE	-0.0325
	KLEE-PRUNE-MERGE	0.0530
link	KLEE-MERGE-5	-0.0228
	KLEE-MERGE-20	0.0546
	KLEE-NO-FF-MERGE-5	-0.0274
	KLEE-NO-FF-MERGE-20	-0.0365
	KLEE-PRUNE	-0.0232
	KLEE-PRUNE-MERGE	0.0099
ln	KLEE-MERGE-5	0.0184
	KLEE-MERGE-20	0.0138
	KLEE-NO-FF-MERGE-5	-0.0311
	KLEE-NO-FF-MERGE-20	-0.0195
	KLEE-PRUNE	-0.0269
	KLEE-PRUNE-MERGE	-0.0070
logname	KLEE-MERGE-5	0.0260
	KLEE-MERGE-20	0.0064
	KLEE-NO-FF-MERGE-5	0.0228
	KLEE-NO-FF-MERGE-20	0.0280
	KLEE-PRUNE	-0.0398
	KLEE-PRUNE-MERGE	0.0312
ls	KLEE-MERGE-5	0.0458
	KLEE-MERGE-20	0.0083
	KLEE-NO-FF-MERGE-5	0.0072
	KLEE-NO-FF-MERGE-20	-0.0529
	KLEE-PRUNE	-0.1068
	KLEE-PRUNE-MERGE	-0.1165
mkdir	KLEE-MERGE-5	0.0171
	KLEE-MERGE-20	0.0273
	KLEE-NO-FF-MERGE-5	0.0090
	KLEE-NO-FF-MERGE-20	0.0024
	KLEE-PRUNE	0.0148
	KLEE-PRUNE-MERGE	-0.0153
mkfifo	KLEE-MERGE-5	-0.0101
	KLEE-MERGE-20	-0.0881
	KLEE-NO-FF-MERGE-5	0.0174
	KLEE-NO-FF-MERGE-20	-0.0254
	KLEE-PRUNE	-0.0264
	KLEE-PRUNE-MERGE	-0.0090
mknod	KLEE-MERGE-5	0.0804
	KLEE-MERGE-20	0.0410
	KLEE-NO-FF-MERGE-5	0.0514
	KLEE-NO-FF-MERGE-20	0.0825
	KLEE-PRUNE	0.0212
	KLEE-PRUNE-MERGE	0.0145

UTILITY	ALGORITHM	COVERAGE (%) DIFFERENCE — ALGORITHM AND KLEE_BASE
mktemp	KLEE-MERGE-5	0.0887
	KLEE-MERGE-20	-0.0665
	KLEE-NO-FF-MERGE-5	-0.0201
	KLEE-NO-FF-MERGE-20	-0.0266
	KLEE-PRUNE	-0.0680
	KLEE-PRUNE-MERGE	0.0518
mv	KLEE-MERGE-5	-0.1497
	KLEE-MERGE-20	-0.1440
	KLEE-NO-FF-MERGE-5	-0.1257
	KLEE-NO-FF-MERGE-20	-0.1360
	KLEE-PRUNE	-0.1214
	KLEE-PRUNE-MERGE	-0.2088
nice	KLEE-MERGE-5	0.0573
	KLEE-MERGE-20	0.0692
	KLEE-NO-FF-MERGE-5	0.0782
	KLEE-NO-FF-MERGE-20	0.0737
	KLEE-PRUNE	0.0711
	KLEE-PRUNE-MERGE	0.1428
nohup	KLEE-MERGE-5	0.1200
	KLEE-MERGE-20	0.1203
	KLEE-NO-FF-MERGE-5	0.1223
	KLEE-NO-FF-MERGE-20	0.0072
	KLEE-PRUNE	0.1113
	KLEE-PRUNE-MERGE	0.1264
od	KLEE-MERGE-5	-0.0080
	KLEE-MERGE-20	-0.0087
	KLEE-NO-FF-MERGE-5	0.0080
	KLEE-NO-FF-MERGE-20	0.0020
	KLEE-PRUNE	0.0167
	KLEE-PRUNE-MERGE	0.0933
paste	KLEE-MERGE-5	0.2022
	KLEE-MERGE-20	0.1604
	KLEE-NO-FF-MERGE-5	0.2114
	KLEE-NO-FF-MERGE-20	0.2061
	KLEE-PRUNE	0.2716
	KLEE-PRUNE-MERGE	0.2343
pathchk	KLEE-MERGE-5	0.0484
	KLEE-MERGE-20	0.0471
	KLEE-NO-FF-MERGE-5	0.0574
	KLEE-NO-FF-MERGE-20	0.0409
	KLEE-PRUNE	0.0549
	KLEE-PRUNE-MERGE	0.0379
pinky	KLEE-MERGE-5	-0.0129
	KLEE-MERGE-20	-0.0266
	KLEE-NO-FF-MERGE-5	-0.0099
	KLEE-NO-FF-MERGE-20	-0.0345
	KLEE-PRUNE	-0.0190
	KLEE-PRUNE-MERGE	-0.0516

UTILITY	ALGORITHM	COVERAGE (%) DIFFERENCE — ALGORITHM AND KLEE_BASE
pr	KLEE-MERGE-5	-0.0781
	KLEE-MERGE-20	-0.1295
	KLEE-NO-FF-MERGE-5	-0.0969
	KLEE-NO-FF-MERGE-20	-0.0431
	KLEE-PRUNE	-0.0998
	KLEE-PRUNE-MERGE	-0.0925
printf	KLEE-MERGE-5	-0.1034
	KLEE-MERGE-20	-0.0982
	KLEE-NO-FF-MERGE-5	-0.0735
	KLEE-NO-FF-MERGE-20	-0.0837
	KLEE-PRUNE	-0.0538
	KLEE-PRUNE-MERGE	-0.0810
ptx	KLEE-MERGE-5	-0.1504
	KLEE-MERGE-20	-0.1506
	KLEE-NO-FF-MERGE-5	-0.1317
	KLEE-NO-FF-MERGE-20	-0.1322
	KLEE-PRUNE	-0.1331
	KLEE-PRUNE-MERGE	-0.1488
readlink	KLEE-MERGE-5	0.0883
	KLEE-MERGE-20	0.0470
	KLEE-NO-FF-MERGE-5	0.0165
	KLEE-NO-FF-MERGE-20	0.0555
	KLEE-PRUNE	0.0882
	KLEE-PRUNE-MERGE	0.0211
rm	KLEE-MERGE-5	0.0226
	KLEE-MERGE-20	-0.0266
	KLEE-NO-FF-MERGE-5	-0.0317
	KLEE-NO-FF-MERGE-20	-0.0542
	KLEE-PRUNE	-0.0326
	KLEE-PRUNE-MERGE	-0.0215
rmdir	KLEE-MERGE-5	0.0105
	KLEE-MERGE-20	0.0213
	KLEE-NO-FF-MERGE-5	-0.0070
	KLEE-NO-FF-MERGE-20	-0.0197
	KLEE-PRUNE	-0.0075
	KLEE-PRUNE-MERGE	0.0185
seq	KLEE-MERGE-5	0.0181
	KLEE-MERGE-20	-0.0210
	KLEE-NO-FF-MERGE-5	-0.0685
	KLEE-NO-FF-MERGE-20	-0.0497
	KLEE-PRUNE	0.0288
	KLEE-PRUNE-MERGE	-0.0071
setuidgid	KLEE-MERGE-5	0.0595
	KLEE-MERGE-20	-0.0714
	KLEE-NO-FF-MERGE-5	0.0231
	KLEE-NO-FF-MERGE-20	0.0284
	KLEE-PRUNE	-0.0287
	KLEE-PRUNE-MERGE	0.0047

UTILITY	ALGORITHM	COVERAGE (%) DIFFERENCE — ALGORITHM AND KLEE_BASE
shred	KLEE-MERGE-5	-0.0350
	KLEE-MERGE-20	0.0725
	KLEE-NO-FF-MERGE-5	-0.0225
	KLEE-NO-FF-MERGE-20	-0.0054
	KLEE-PRUNE	0.0029
	KLEE-PRUNE-MERGE	0.0778
shuf	KLEE-MERGE-5	-0.0512
	KLEE-MERGE-20	-0.1405
	KLEE-NO-FF-MERGE-5	-0.0771
	KLEE-NO-FF-MERGE-20	-0.1005
	KLEE-PRUNE	-0.1849
	KLEE-PRUNE-MERGE	-0.0584
sleep	KLEE-MERGE-5	0.0917
	KLEE-MERGE-20	-0.0011
	KLEE-NO-FF-MERGE-5	0.0589
	KLEE-NO-FF-MERGE-20	0.0361
	KLEE-PRUNE	0.0129
	KLEE-PRUNE-MERGE	0.1081
split	KLEE-MERGE-5	0.0883
	KLEE-MERGE-20	0.1047
	KLEE-NO-FF-MERGE-5	0.0197
	KLEE-NO-FF-MERGE-20	0.0187
	KLEE-PRUNE	0.0329
	KLEE-PRUNE-MERGE	0.0879
stty	KLEE-MERGE-5	-0.0632
	KLEE-MERGE-20	-0.0420
	KLEE-NO-FF-MERGE-5	-0.0482
	KLEE-NO-FF-MERGE-20	-0.0916
	KLEE-PRUNE	-0.0559
	KLEE-PRUNE-MERGE	-0.1565
tail	KLEE-MERGE-5	-0.0894
	KLEE-MERGE-20	-0.0611
	KLEE-NO-FF-MERGE-5	-0.0455
	KLEE-NO-FF-MERGE-20	-0.0570
	KLEE-PRUNE	0.0081
	KLEE-PRUNE-MERGE	-0.1044
test	KLEE-MERGE-5	0.1028
	KLEE-MERGE-20	0.0768
	KLEE-NO-FF-MERGE-5	0.1224
	KLEE-NO-FF-MERGE-20	0.1066
	KLEE-PRUNE	0.0482
	KLEE-PRUNE-MERGE	0.0706
touch	KLEE-MERGE-5	-0.0819
	KLEE-MERGE-20	-0.0629
	KLEE-NO-FF-MERGE-5	-0.0910
	KLEE-NO-FF-MERGE-20	-0.0917
	KLEE-PRUNE	-0.0547
	KLEE-PRUNE-MERGE	-0.0678

UTILITY	ALGORITHM	COVERAGE (%) DIFFERENCE — ALGORITHM AND KLEE_BASE
tr	KLEE-MERGE-5	0.0622
	KLEE-MERGE-20	0.0316
	KLEE-NO-FF-MERGE-5	-0.0255
	KLEE-NO-FF-MERGE-20	0.0081
	KLEE-PRUNE	0.0067
	KLEE-PRUNE-MERGE	-0.0547
tsort	KLEE-MERGE-5	-0.0356
	KLEE-MERGE-20	-0.0750
	KLEE-NO-FF-MERGE-5	0.0101
	KLEE-NO-FF-MERGE-20	-0.0018
	KLEE-PRUNE	-0.0059
	KLEE-PRUNE-MERGE	-0.0402
tty	KLEE-MERGE-5	0.0935
	KLEE-MERGE-20	0.0754
	KLEE-NO-FF-MERGE-5	0.0758
	KLEE-NO-FF-MERGE-20	0.0370
	KLEE-PRUNE	0.0543
	KLEE-PRUNE-MERGE	0.1015
uname	KLEE-MERGE-5	0.0844
	KLEE-MERGE-20	0.1163
	KLEE-NO-FF-MERGE-5	0.0866
	KLEE-NO-FF-MERGE-20	0.0834
	KLEE-PRUNE	0.0874
	KLEE-PRUNE-MERGE	0.1080
unexpand	KLEE-MERGE-5	-0.0701
	KLEE-MERGE-20	-0.0063
	KLEE-NO-FF-MERGE-5	-0.0890
	KLEE-NO-FF-MERGE-20	-0.0997
	KLEE-PRUNE	0.0315
	KLEE-PRUNE-MERGE	-0.0786
uniq	KLEE-MERGE-5	-0.0136
	KLEE-MERGE-20	0.0093
	KLEE-NO-FF-MERGE-5	0.0177
	KLEE-NO-FF-MERGE-20	-0.0734
	KLEE-PRUNE	-0.0246
	KLEE-PRUNE-MERGE	-0.0678
unlink	KLEE-MERGE-5	0.0992
	KLEE-MERGE-20	0.1239
	KLEE-NO-FF-MERGE-5	0.0752
	KLEE-NO-FF-MERGE-20	0.0753
	KLEE-PRUNE	0.1085
	KLEE-PRUNE-MERGE	0.0974
wc	KLEE-MERGE-5	-0.0238
	KLEE-MERGE-20	0.0256
	KLEE-NO-FF-MERGE-5	-0.0774
	KLEE-NO-FF-MERGE-20	0.0286
	KLEE-PRUNE	0.1253
	KLEE-PRUNE-MERGE	0.0669

UTILITY	ALGORITHM	COVERAGE (%) DIFFERENCE — ALGORITHM AND KLEE_BASE
who	KLEE-MERGE-5	0.0389
	KLEE-MERGE-20	-0.0024
	KLEE-NO-FF-MERGE-5	-0.0082
	KLEE-NO-FF-MERGE-20	-0.0073
	KLEE-PRUNE	0.0292
	KLEE-PRUNE-MERGE	-0.0213
whoami	KLEE-MERGE-5	-0.0014
	KLEE-MERGE-20	0.0203
	KLEE-NO-FF-MERGE-5	0.0129
	KLEE-NO-FF-MERGE-20	0.0163
	KLEE-PRUNE	-0.0204
	KLEE-PRUNE-MERGE	0.0159

Appendix B: LOC and #include count for all COREUTILS

UTIL	LOC	#include	UTIL	LOC	#include	UTIL	LOC	#include
base64	236	10	id	297	13	rm	272	14
chcon	437	13	join	714	13	rmdir	150	7
chgrp	243	13	kill	293	8	seq	319	9
chmod	400	14	link	62	8	setuidgid	172	12
chown	252	11	ln	427	14	shred	790	15
cksum	224	5	logname	58	8	shuf	329	11
comm	194	10	ls	3247	38	sleep	105	11
cp	818	17	mkdir	159	12	split	437	14
csplit	1085	13	mkfifo	107	9	stty	1630	10
cut	617	11	mknod	183	10	tail	1232	20
date	459	13	mktemp	230	10	test	642	10
dd	1278	13	mv	358	16	touch	325	15
df	708	14	nice	139	9	tr	1314	10
dircolors	393	11	nohup	156	13	tsort	370	10
du	709	20	od	1394	10	tty	82	7
env	103	7	paste	338	6	uname	283	9
expand	295	8	pathchk	302	8	unexpand	365	8
factor	151	14	pinky	453	11	uniq	421	14
fmt	646	8	pr	1712	13	unlink	57	8
fold	234	8	printf	537	11	wc	542	12
head	760	12	ptx	1336	12	who	608	11

Bibliography

[1] C. S. Păsăreanu and W. Visser, "A survey of new trends in symbolic execution for software testing and analysis," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 11, no. 4, pp. 339–353, 2009. DOI: 10.1007/s10009-009-0118-1

[2] P. Godefroid, M. Y. Levin, and D. Molnar, "Automated whitebox fuzz testing," in *Proceedings of Network and Distributed Systems Security (NDSS)*, 2008, pp. 151–166.

[3] "Coreutils," http://www.gnu.org/software/coreutils.

[4] Common Vulnerabilities and Exposures, "CVE-2014-0160," http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160.

[5] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2005, pp. 213–223. DOI: 10.1145/1065010.1065036

[6] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically generating inputs of death," *ACM Transactions on Information and System Security (TISSEC)*, vol. 12, no. 2, pp. 1–38, 2008. DOI: 10.1145/1455518.1455522

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

[8] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice: preliminary assessment," in *Proceedings of the 33rd International Conference on Software Engineering*, New York, NY, USA, 2011, pp. 1066–1071. DOI: 10.1145/1985793.1985995

[9] W. Kimball, "On model checking of binary programs," Ph.D. dissertation, Air Force Institute of Technology (AFIT), 2013.

[10] "The KLEE symbolic virtual machine," http://klee.github.io/klee/.

[11] "llvm-gcc - LLVM C front-end," http://llvm.org/releases/2.9/docs/CommandGuide/html/llvmgcc.html.

[12] "clang: a C language family frontend for LLVM," http://clang.llvm.org/.

[13] "High level overview of KLEE," http://www.doc.ic.ac.uk/~dsl11/klee-doxygen/overview.html.

[14] "gcov: a test coverage program," http://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[15] T. Xie, N. Tillmann, P. de Halleux, and W. Schulte, "Fitness-guided path exploration in dynamic symbolic execution," in *Proceedings the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2009)*, 2009, pp. 359–368.

[16] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, New York, NY, USA, 2005, pp. 263–272. DOI: 10.1145/1081706.1081750

[17] X. Xiao, X. Zhang, and X. Li, "New approach to path explosion problem of symbolic execution," in *Proceedings of the 2010 First International Conference on Pervasive Computing, Signal Processing and Applications*, Washington, DC, USA, 2010, pp. 301–304. DOI: 10.1109/PCSPA.2010.80

[18] T. Hansen, P. Schachte, and H. Søndergaard, "State Joining and Splitting for the Symbolic Execution of Binaries," in *Runtime Verification*, S. Bensalem and D. A. Peled, Eds. Berlin, Heidelberg: Springer-Verlag, 2009, vol. 5779, ch. State Joining and Splitting for the Symbolic Execution of Binaries, pp. 76–92.

[19] P. Boonstoppel, C. Cadar, and D. Engler, "RWset: attacking path explosion in constraint-based test generation," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 351–366.

[20] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, 2012, pp. 193–204. DOI: 10.1145/2254064.2254088

[21] S. Bugrara and D. Engler, "Redundant state detection for dynamic symbolic execution," in *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, San Jose, CA, 2013, pp. 199–211.

[22] R. Majumdar and K. Sen, "Hybrid concolic testing," in *Proceedings of the 29th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 416–426. DOI: 10.1109/ICSE.2007.41

[23] N. Tillmann and J. De Halleux, "Pex: White box test generation for .NET," in *Proceedings of the 2nd International Conference on Tests and Proofs*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 134–153.

[24] P. Godefroid, “Compositional dynamic test generation,” in *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Ssymposium on Principles of Programming Languages*. New York, NY, USA: ACM, 2007, pp. 47–54. DOI: 10.1145/1190216.1190226

[25] Z. Wan and B. Zhou, “Effective code coverage in compositional systematic dynamic testing,” in *Information Technology and Artificial Intelligence Conference (ITAIC), 2011 6th IEEE Joint International*, vol. 1, 2011, pp. 173 –176. DOI: 10.1109/ITAIC.2011.6030179

[26] R. Santelices and M. J. Harrold, “Exploiting program dependencies for scalable multiple-path symbolic execution,” in *Proceedings of the 19th International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM, 2010, pp. 195–206. DOI: 10.1145/1831708.1831733

[27] M. Staats and C. Păsăreanu, “Parallel symbolic execution for structural test generation,” in *Proceedings of the 19th International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM, 2010, pp. 183–194. DOI: 10.1145/1831708.1831732

[28] P. Godefroid, M. Y. Levin, and D. Molnar, “SAGE: whitebox fuzzing for security testing,” *Queue*, vol. 10, no. 1, pp. 20–27, 2012. DOI: 10.1145/2090147.2094081

[29] W. Visser, C. S. Păsăreanu, and S. Khurshid, “Test input generation with Java PathFinder,” in *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, New York, NY, USA, 2004, pp. 97–107. DOI: 10.1145/1007512.1007526

[30] “Coreutils experiments,” http://klee.github.io/klee/CoreutilsExperiments.html.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)* 19–06–2014	**2. REPORT TYPE** Master's Thesis	**3. DATES COVERED** *(From — To)* Oct 2013–Jun 2014

4. TITLE AND SUBTITLE Using State Merging and State Pruning to Address the Path Explosion Problem Faced by Symbolic Execution	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Copeland, Patrick T., Civilian	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 Hobson Way WPAFB, OH 45433-7765	**8. PERFORMING ORGANIZATION REPORT NUMBER** AFIT-ENG-T-14-J-3
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Intentionally left blank	**10. SPONSOR/MONITOR'S ACRONYM(S)**
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

13. SUPPLEMENTARY NOTES

This work is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT

Symbolic execution is a promising technique to discover software vulnerabilities and improve the quality of code. However, symbolic execution suffers from a path explosion problem where the number of possible paths within a program grows exponentially with respect to loops and conditionals. New techniques are needed to address the path explosion problem. This research presents a novel algorithm which combines the previously researched techniques of state merging and state pruning. A prototype of the algorithm along with a pure state merging and pure state pruning are implemented in the KLEE symbolic execution tool with the goal of increasing the code coverage. Each algorithm is tested over 66 of the GNU COREUTILS utilities. State merging combined with state pruning outperforms the unmodified version of KLEE on 53% of the COREUTILS. These results confirm that state merging with pruning has viability in addressing the path explosion problem of symbolic execution.

15. SUBJECT TERMS

symbolic execution, state merging, state pruning, path explosion, program analysis

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. Gilbert L. Peterson (ENG)
					19b. TELEPHONE NUMBER *(include area code)*
U	U	U	UU	89	(937) 255-3636 x4281 gilbert.peterson@afit.edu