Copyright
by
Sarmad Makhdoom
2014

# LAHORE UNIVERSITY OF MANAGEMENT SCIENCES

**Department of Computer Science**

**CERTIFICATE**

***I hereby recommend that the thesis prepared under my supervision by:***

**Sarmad Makhdoom**

***on Title:*** **Incremental Symbolic Execution**

***be accepted in partial fulfillment of the requirements for the MS degree.***

**Advisor (Chairperson of Defense Committee)**

***Recommendation of Thesis Defense Committee:***

(at least a Supervisor and one more member is required as per rule)

Name Signature Date

Name Signature Date

# Incremental Symbolic Execution

by

**Sarmad Makhdoom**

A dissertation submitted in partial fulfillment of the requirements
for the degree of Master of Science in the School of Science and Engineering
at the Lahore University of Management Sciences,
Lahore, Pakistan

Lahore Univeristy of Management and Sciences
2014

# Acknowledgments

Foremost, I would like to express my sincere gratitude to my advisor Dr Junaid Haroon Siddiqui for the continuous support of my MS thesis, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for this thesis. Besides my advisor, I would like to thank Dr Arif Zaman for their insightful comments and being member of my thesis committee.

My friends at LUMS Kamran Khalil, Qasim Khalid, Ahmad Mujtaba my roommate, and Hassan Mustafa gave me a great time here. Hassan is the person who motivated me for doing this research otherwise I would be doing my masters by studying unexciting courses. Time at LUMS is the finest time of my life when I have met with great friends, great teachers and very encouraging study environment.

Most importantly, I would like to thank my fiancé Shafaq Khalid for supporting me throughout my masters and always giving me encouragement, support and obviously time to finish this study in timely manner.

In the end, I would like to thank my mother, and siblings, who have all been very supportive during my stay at Lahore for bachelors and masters. I look forward to see them all.

# ABSTRACT

Scaling software analysis techniques based on source-code, such as symbolic execution and data flow analyses, remains a challenging problem for systematically checking software systems. In this work, we aim to efficiently apply symbolic execution in increments based on versions of code. Our technique is based entirely on dynamic analysis and patches completely automated test suites based on the code changes. Our key insight is that we can eliminate constraint solving for unchanged code by checking constraints using the test suite of a previous version. Checking constraints is orders of magnitude faster than solving them. This is in contrast to previous techniques that rely on inexact static analysis or cache of previously solved constraints. Our technique identifies ranges of paths, each bounded by two concrete tests from the previous test suite. Exploring these path ranges covers all paths affected by code changes up to a given depth bound. We show that complete bounded symbolic execution on an earlier version of code combined with incremental symbolic execution with our proposed technique provides equal coverage and bug-finding ability to the complete bounded symbolic execution of modified code. Our experiments show that incremental symbolic execution based on dynamic analysis is an order of magnitude faster than running complete standard symbolic execution on the new version of code.

# Table of Contents

**Acknowledgments** **i**

**Abstract** **ii**

**List of Tables** **v**

**List of Figures** **vi**

**Chapter 1. Introduction** **1**

1.1 Problem Statement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

1.2 Thesis and Proposed Technique . . . . . . . . . . . . . . . . . . . . . . . 2

1.3 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

**Chapter 2. Illustrative Overview** **5**

**Chapter 3. Technique** **9**

3.1 Incremental Symbolic Execution . . . . . . . . . . . . . . . . . . . . . . 9

3.2 Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

3.3 Infeasible space becomes feasible . . . . . . . . . . . . . . . . . . . . . . 14

**Chapter 4. Implementation** **16**

**Chapter 5. Evaluation** **18**

5.1 Methodology . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19

5.2 Experimental Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19

5.3 Discussion on results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

**Chapter 6. Related Work 32**

6.1 Incremental Symbolic Execution based on Static Analysis . . . . . . . . . . . 33

6.2 Incremental Symbolic Execution based on caching solutions to path constraints 34

6.3 Other ways to scale symbolic execution . . . . . . . . . . . . . . . . . . . . . 35

**Chapter 7. Conclusion 37**

**Bibliography 38**

**Vita 42**

# List of Tables

5.1 Symbolic execution results for 85 program from GNU Coreutils suite of Unix utilities of version 7.1, 7.2 (with and without incremental) . . . . . . . 24

5.2 Symbolic execution results for 85 program from GNU Coreutils suite of Unix utilities of version 8.1 (with and without incremental) . . . . . . . . . 28

# List of Figures

2.1 Execution tree of program 1 . . . . . . . . . . . . . . . . . . . . . . . . . 6

2.2 Execution tree of program 2 . . . . . . . . . . . . . . . . . . . . . . . . . 7

3.1 Execution tree of new program without execution . . . . . . . . . . . . . . 12

3.2 Execution tree of new program with one valid test reaching the leaf . . . . 13

3.3 Range for modified execution tree with two bounded test cases . . . . . . . 13

3.4 Execution tree of program bounded by two valid test cases having infeasible area . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

5.1 Execution time of 85 program from GNU Coreutils suite version 7.2 using incremental and non-incremental approach. Square boxes show the time for symbolic execution time by using incremental testing providing seeds from version 7.1. Circle shows the execution time of version 7.2 without using incremental technique. . . . . . . . . . . . . . . . . . . . . . . . . . . 21

5.2 Execution time of 85 program from GNU Coreutils suite version 8.1 using incremental and non-incremental approach. Square boxes show the time for symbolic execution time by using incremental testing providing seeds from version 7.2. Circle shows the execution time of version 8.1 without using incremental technique. . . . . . . . . . . . . . . . . . . . . . . . . . . 22

# Chapter 1

# INTRODUCTION

In three decades time, symbolic execution has come up as a powerfully effective tool as program analysis tool that is based on the exploration of program paths [8, 16]. For the symbolic execution *path conditions* are to be built —given a path, then this path represents an input variable constraint. One of the applications of symbolic execution is to generate test inputs for the enhancement of code coverage. In order to automate the symbolic execution, solvers of constraints are required; in some techniques decision procedures are used instead of solvers [3, 10] that control the constraints class in the ensuing path conditions.

When it comes to constraint solving technology, a lot of change has been seen in the last constraint. In this regard SAT [26] and SMT [3, 10] are worth mentioning. In order to decrease the complexity of formula solving in the real application, raw computation has evolved with efficient results. These innovation have opened arenas for more detailed researches in this regard, as in the present era handling constraints of modern programing and traditional analyses like test input generation [15, 13, 21, 5] are not the only things that can be handled effectively but the non-conventional ones are also being done like program equivalence assessment [20], for the modification of data structures for error recovery [11] and for the approximation of the consumption of power [22].

## 1.1 Problem Statement

Despite the advances in the field of symbolic execution, a basic limiting factor is its complex path-based analysis. Due to this complex analysis, symbolic execution takes significant time and its use becomes impractical for integrating in development work flow. A key observation, however, is that in a development work flow, code changes in increments and the increments are often very small. Numerous latest research projects have worked and implemented on this topic. Two approaches have been used. One approach is to use static analysis to find differences in the code flow graph (CFG) and either prune symbolic execution by eliminating paths that will not be changed for sure [19] or use heuristics to guide a manual test suite towards code changes [18]. The other approach is based on caching previous results of symbolic analysis so that future runs are faster [29, 28]. The later technique does not utilize the information about code changes directly and relies on the fact that similar code will result in similar path conditions and hence the solving can greatly benefit from caching prior results.

## 1.2 Thesis and Proposed Technique

This thesis presents a novel technique for incremental symbolic execution using a pure dynamic analysis approach without caching any prior results. Our key insight is that while solving path conditions is costly, generating and checking them is orders of magnitude faster. Instead of performing an inexact static analysis to eliminate generation of path conditions, we generate path conditions and use the prior test suite to check satisfiability. The tests are kept in memory and efficiently divided along different paths of exploration to check against the generated path condition. Storing concrete tests and checking their satisfiability is much more efficient than caching path conditions, canonicalizing them, and comparing them.

Paths in changed code do not match any prior tests and have to be fully explored. The first and last such path delineate a *range* that can be explored. This builds on our previous work on *ranged symbolic execution* [24] that introduced the concept of ranges of paths to be explored and used it to enable parallel symbolic execution.

The basic idea behind the ranged symbolic execution is that *state* of a symbolic execution can be determined by a *test input*. It states that if we introduce some sort of ordering in the execution like always execute true branch first followed by the false branch for each non-deterministic branching statement in the program during the exploration, we can pause and resume the symbolic execution. Such mechanism is already implemented in common symbolic execution [15, 5, 2], by doing so we can always partition the execution tree into two different sets with the help of two test inputs; One consists of *explored* paths while the other set consists of *unexplored* paths. Moreover, among the test inputs a *linear ordering* can be defined if we follow order as discussed above in the exploration. These two test inputs should not execute the same path or should not lead the program to an infinite loop. The ordering defines which path in the program should execute first during the symbolic execution. So a well-ordered pair of tests, lets say $\langle \tau, \tau' \rangle$, expresses a *range* of two paths that bound some space in the execution tree $[\rho_1, \ldots, \rho_k]$ in which a path $\rho_1$ is taken by $\tau$ and path $\rho_k$ is taken by $\tau'$, and for $1 \leq i < k$, path $\rho_{i+1}$ is executed after $\rho_i$.

The encoding allows us to separate out the ranges affected by code changes for incremental symbolic execution. We generate and solve path conditions for all test cases in this range and generate new tests in the test suite. For all paths that have a matching test case from prior version, we copy the old test. Any tests that no longer have a matching path get discarded automatically. This enables a totally automated maintenance of symbolic execution generated test suite.

## 1.3 Contributions

We make the following contributions:

- **Dynamic analysis for unchanged path identification.** We introduce a pure dynamic analysis based approach that identifies paths unaffected by code changes by using efficient constraint checking using a prior test suite.
- **Identification of affected path ranges.** We identify each range of affected paths and based on our previous work on ranged symbolic execution [24], represent it succinctly and efficiently using only two concrete tests.
- **Incremental symbolic execution.** Using dynamic analysis to identify unchanged paths and identifying changed paths in ranges, we are able to use ranged symbolic execution to explore the affected ranges completely up to the given depth bound. This enables very efficient incremental symbolic execution without any dependence on inexact static analysis and without caching complicated path conditions.
- **Implementation.** We implemented incremental symbolic execution on top of our ranged symbolic execution [24] which is in turn built upon KLEE [5]. KLEE is an open-source tool for symbolic execution, which works on LLVM [1] is a transitional/intermediate language of compiler which is just like assembly language and it has only branches that either evaluated true or false. But it has much more information as compared to assembly language.
- **Evaluation.** We evaluated incremental symbolic execution using 85 programs in the GNU Coreutils —the broadly used set of utilities in Unix environment. We observed that incremental symbolic execution is 73% faster and we are saving 80% solver time by giving 67% less queries.

# Chapter 2

# ILLUSTRATIVE OVERVIEW

Symbolic execution (forward) is used for the execution of symbolic values [9, 16]. Symbolic execution constitute of two paradigms; defining the semantics of operations and secondly to maintain a *path condition*. A path condition defines necessary constraints on input variables that are need to satisfied in order to execute the related path.

```
1  int middle(int a, int b, int z) {
2    if (a<b) {
3      if (b<z) return b;
4      else if (a<z) return z;
5      else return a;
6    } else if (a<z) return a;
7    else if (b<z) return z;
8    else return b; }
```

Program 1: To find middle of three integers

The program 1 has a function that selects the middle integer values from the three integers. The execution tree for such a program is illustrated in figure 2.1 and the 6 path conditions are depicted in it. For the solution of the path conditions for each of the paths, off shelf SAT solvers are used that are meant for concrete tests. For instance, for path 2 the solution can be, A=1, B=3 and C=3. In this way values can be found for the 6 path conditions. The whole program is done symbolically and then a test suite in generated for this particular version of code. The tests are comprised of discreet values and take the program to those paths.

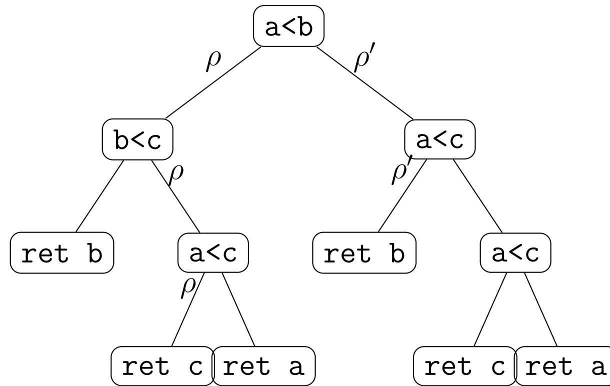


Figure 2.1: Execution tree of program 1

In order to execute the program symbolically, the integer inputs behavior is taken into consideration for example, `A`, `B` and `C`. While making conditional statements both the possible outcomes are considered. The operations on symbols are operated algebraically.

Six paths are explored during the symbolic execution of program `middle`:

```
path 1: [A < B < C] L2 -> L3
path 2: [A < C < B] L2 -> L3 -> L4
path 3: [C < A < B] L2 -> L3 -> L4 -> L5
path 4: [B < A < C] L2 -> L6
path 5: [B < C < A] L2 -> L6 -> L7
path 6: [C < B < A] L2 -> L6 -> L7 -> L8
```

Program 2 shows a newer version of the program. Let's assume line numbers 4 and 8 are changed in this version of the program. This results in the new execution tree shown in Figure 2.2. Here we can see that the path conditions are also changed. We need to identify the change in the execution tree in order to perform incremental symbolic execution. We will use our previously generated test suite to find the difference in the execution tree.

```
1  int middle(int a, int b, int c) {
2    if (a<b) {
3      if (b<c) return b;
4      else if (a<=c) return c;
5      else return a;
6    } else if (a<c) return a;
7    else if (b<c) return c;
8    else return a; }
```

Program 2: Modified version of mid shown in Program 1

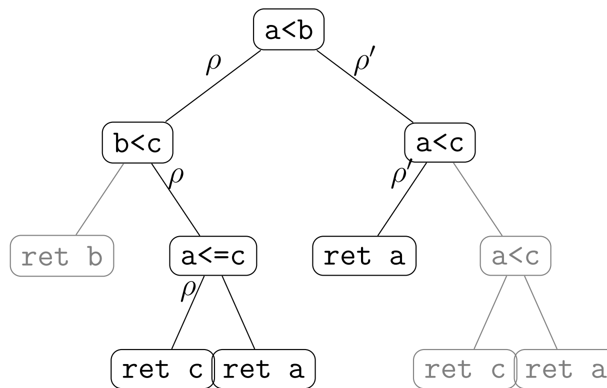


Figure 2.2: Execution tree of program 2

We start by analyzing the modified tree while maintaining a list of all tests satisfying the current paths. We explore the first path that will take left most path in the execution tree. We build the path condition as usual and check against satisfying tests. Since this path is unchanged one test from previous suite will be attached to this path and we can output this test without using any solver calls. Any branches that are unsatisfied by any test from previous suite are discarded. We later show that no new valid path can appear between two valid tests from the previous suite. The second test input will not remain attached to the second path. The program will take some path but it won't be the one we discovered last time. The path condition would be different this time and no test from previous suite will validate with it. We now know a point where path are starting to diverge. This is the starting point of our range that we need to reexamine for the changes.

We then keep on exploring paths and finding test cases from previous suite that satisfy them until we come to some situation where an old test take the same exact path generating the old path condition in the program. In this example that would be the path condition 6 where program will take exact same path again and will validates the path condition as well. At this point, we mark the previous test case as our last point in the range that is affected by code modification. Now after this analysis we have a range that states that the program is changed from this path condition to next path condition. We will complete symbolic execution only on the area identified in the program that will results in 3 new tests. These tests are those, which were invalidated, in the newer version of the program and results in the new path conditions. We then use the SAT solver to evaluate them and add these new values to our test suite. We can safely remove our old test inputs that are invalided in the next run and reuse that are still valid.

# Chapter 3

# TECHNIQUE

In this section we will discuss our approach to dynamically figure out the areas in the execution tree that is modified in the newer version of code. We will first discuss the key idea behind the incremental symbolic testing procedure (Section 3.1), then discuss our algorithm for handling ranges by giving test inputs (Section 3.2), then discuss why it is not possible to have feasible search space between two valid test cases in the execution tree (Section 3.3) and last will talk about implementation part of our technique (Section 3.4). This thesis assumes a standard bounded depth-first symbolic execution where path exploration is systematic and for each condition, the 'true' branch is explored before the 'false' branch. Such exploration is standard in commonly used symbolic execution techniques, such as generalized symbolic execution using JPF [15], CUTE [21], and KLEE [5].

## 3.1 Incremental Symbolic Execution

Whenever we execute a program symbolically, the interpreter generates a symbolic execution tree characterizing the execution paths followed by the symbolic execution. In this tree it associates a directed arc connecting the associated nodes with each transition between the statements. For every IF statement execution, the associated node has two arcs leaving the node, which represents the TRUE and FALSE part respectively. For every concrete input the program takes a path in that execution tree.

When we execute a program symbolically, we explore its execution tree and maintains the state of program in the form of path-conditions. When the executor reaches the leaf node, it solves the path condition to get the concrete values. If we supply those concrete values to the program as an input, it will follow the same exact path in the program's execution tree.

It is nature of software engineering that programs tend to change. These changes occur due to bug fixes or addition of new features in the program. Every change in the program code results in the new modified execution tree. In our technique we dynamically figure out the area in the execution tree which is either modified in the new version or it was not present in the previous version of the execution tree.

Testing a program exhaustively requires much time in case of symbolic execution or even with any program testing technique. When new versions are developed, it is obvious understanding that testing those versions should take lesser time because most of the code remains same. But due to the nature of symbolic execution we cannot do that, we have to test the program completely once again that would take as much time as we spent on the base version. In our technique we first symbolically execute the base version of the program, which results in generation of a comprehensive test suite by solving the path conditions. While executing the new version symbolically we use previous test suite in order to produce test suite for the new version. While in the process of exploring states we compare and validate each new path condition with the one we already have in the test suite of base version. If we have a successful comparison, we just add that test case to the new test suite instead of calling the solver. In incremental testing most of the code remains same so is the execution tree. Comparing new path conditions with the old ones is less costly than solving the path condition for the concrete inputs.

## 3.2 Algorithm

Algorithm 1 is used to find and evaluate affected ranges in the program execution tree. It splits test suite on each branch condition in two sets. One set contains all those tests that are satisfiable if that branch evaluated true and other set contains all those test cases that result in false evaluation of branch condition. At this point, if any of the true or false side gets no test cases and all tests go on the other side then that area was infeasible before and we do not need to explore it again. That is first benefit of using our approach in terms of time saving. If any test case is not satisfiable at this point, we start exploring that path with the help of symbolic execution until some other path in the way matches with our previous test inputs.

---

**Algorithm 1:** IncrementalExplore to explore new ranges and not solving the present path conditions in the new program

---

**Input**: A finite set $TestSuite = \{t_1, t_2, \ldots, t_n\}$ of test cases

1 **for** ***each*** *b in BranchCondition* **do**
2     $T_{true} \leftarrow split(TestSuite, b)$; $T_{false} \leftarrow split(TestSuite, \neg b)$
3     $T_{invalid} \leftarrow TestSuite - (T_{true} \cup T_{false})$
4     **if** $T_{invalid} \neq \emptyset$ **then**
5         **if** $T_{true} = \emptyset$ **then**
6             $exploreAndSolve(T_{true})$
7             $IncrementalExplore(T_{false})$
8         **if** $T_{false} = \emptyset$ **then**
9             $exploreAndSolve(T_{false})$
10             $IncrementalExplore(T_{true})$
11     **else**
12         $IncrementalExplore(T_{true})$
13         $IncrementalExplore(T_{false})$

---

We will discuss it with the help of an example. Figure 3.1 shows the execution tree of new version of the program which is unknown to the executor before starting execution. We give test cases to the executor before it start exploring the code. On each branching node

it will split the test suite into two possible subsets where one set belongs to the left side of the tree and other belongs to the right side. On each branch we also validate the test inputs to see whether they are still satisfiable or not.

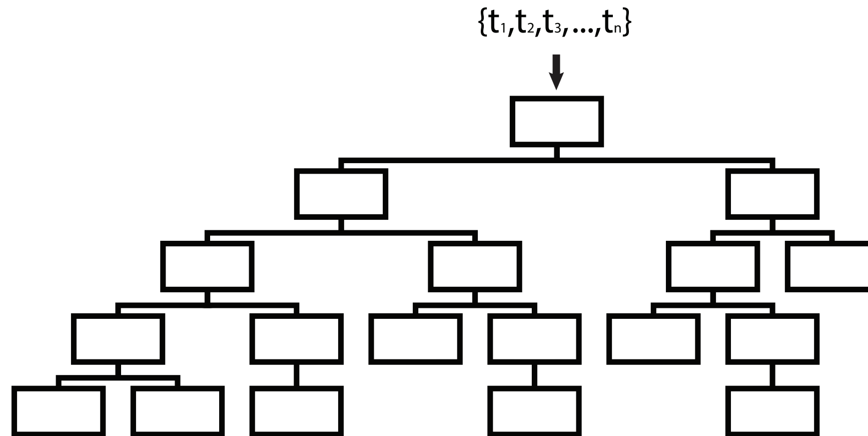


Figure 3.1: Execution tree of new program without execution

In order to understand the concept of finding ranges in the program, lets say one of the test cases reaches the leaf node in the execution tree (as shown in figure 3.2). We compare and validate this test case, and lets say, it turns out to be successfully validated. We then move to the next test case from the test suite, while executing the next test case we discover that next test case is not validating. These validations fail due to change in the execution tree because of change in the path condition. We mark this test case as the start of our range; range which needs to be explored for new test inputs.

We keep on testing the program until all other test cases either fail or we do not find any path in the exploration that matches with the one we already have. Lets say, in figure 3.3, we come across a test case which successfully validates with the current path condition. This is the point where we close our range. So our range starts from the left most test case to the last valid test case as shown in figure 3.3. That range would have all the new states that need to be explored. And any tests present in this area should be solved with the solver for concrete inputs. These tests are added to the new test suite because either they were not

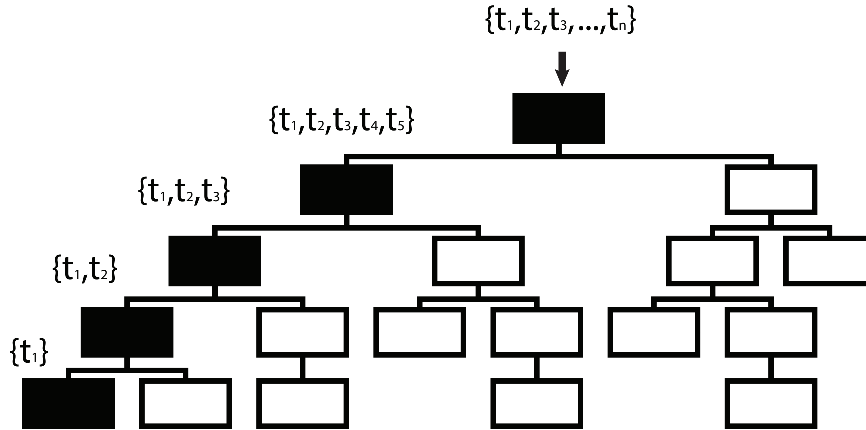


Figure 3.2: Execution tree of new program with one valid test reaching the leaf

present in the old test suite or they were invalidated.

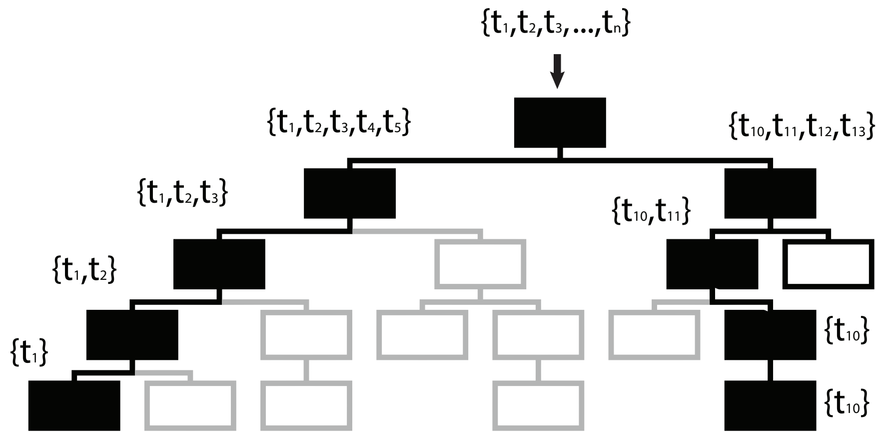


Figure 3.3: Range for modified execution tree with two bounded test cases

We can get such multiple ranges in the execution tree of modified program. As we have already stated that the symbolic execution for incremental testing is in order while exploring the program state space so these ranges would be non-overlapping and we can safely run them in parallel on multiple machines to achieve more speed up for incremental testing.

## 3.3 Infeasible space becomes feasible

Figure 3.4 shows a scenario where some area is infeasible in the first run of the program analysis. It can be any new area that is introduced during the process of change or it is the area that was not solved by the solver in the first run. A valid question to ask is, how our technique behaves if an infeasible area becomes feasible? This infeasible area is bounded by two valid test cases and our algorithm states that there would not be a feasible area in two consecutive valid test cases. It can be explained with the figure easily that this case cannot happen in the reality of the program execution. In order to make infeasible area accessible or solvable now, there must be some change at the node pointed in the figure. Because if that area is feasible now there must be some change in the code that directs the symbolic executor to follow that path in the infeasible area.

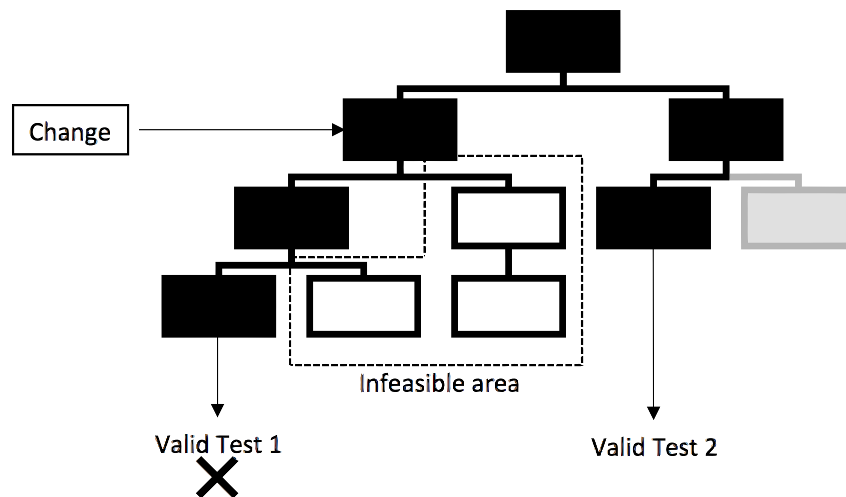


Figure 3.4: Execution tree of program bounded by two valid test cases having infeasible area

Change at node pointed in the figure invalidates test case 1. If there is a change in the path of test case 1, that would not be compared with the one we already have in the previous test suite. To make infeasible area feasible, we must have some invalid test case (test case that was not found in the test suite of base version). So our algorithm would not skip any

area that is affected by the change. Furthermore making test case 1 invalid would be an indication that we are looking for an affected code range. That might the starting point of the range or that invalid test case comes in the middle of any modified range that we are in the process of exploring by our algorithm.

# Chapter 4

# IMPLEMENTATION

As we discussed earlier, the novelty of our technique is to work only on the modified ranges in the program dynamically and solve only those ranges while keeping all those test cases that are valid from the previous test suite. Instead of first finding the ranges and then solving for new test cases, we started exploring for new states as we encounter with some invalid test case.

We modified state-of-the-art tool KLEE [5] that operates on the LLVM [1] bitcode for our implementation of our technique. KLEE takes LLVM bitcode, performs symbolic analysis, and produces test cases along with their path condition files. Test cases generated by the KLEE contains the actual values to use for symbolic data in order to reproduce the path that KLEE followed. These can be used for obtaining code coverage as well as for reproducing the bugs. We used these test inputs, called seeds, generated by KLEE, as an input test suite to the new program.

If we execute KLEE by providing seeds, it stores them in hashmaps. While executing a program symbolically it takes all these seeds side by side. On every branching condition in the program, it divides the seeds for both branches. Any number of seeds that belong to the true side of the branch are associated with it and vice versa. On each step it removes those seeds that are no longer validating with current state of the program. A branch is considered infeasible if all seeds go on the other side.

We took advantage of this seeding system in KLEE to implement our incremental technique. For every branch seeds are compared and divided. If all seeds are validated, it means there is no change in the program. But if any seed fails to validate, it marks our starting point of the range. We start exploring for new states from that point until we find some other seed in the way that was present before and still validating. We modified KLEE so that instead of just checking and moving forward when seeds are supplied, when some seeds fail to validate it starts exploring the program further. We then make new path conditions and solve them in the end for concrete values. These new path conditions and concrete values are added to the new test suite of the program so that they can serve as seeds for the next incremental run.

Comparing and validating the seeds are very easy processes because KLEE uses hashes for seeds as well as the program states, so comparisons would not take much of the computing powers. To validate a seed it just make sure that previous input values for the test case still satisfy the current path condition. This is also not a very CPU intensive job as compared to finding new values by giving path conditions to the solver.

# Chapter 5

# EVALUATION

To evaluate the incremental symbolic execution we used the GNU core utilities[1] - shell, text manipulation and utilites regarding basic file operations for Unix operating system. The utilites are programs between 2000 - 6000 lines of code and fairly medium sized programs. Some of these program performs basic tasks with lot of error checks and thus forms a deep search tree while other performs multiple functions and form a broad search tree. Moreover, because these are included in the every GNU operating systems and they evolve much during the passage of time. We have much iteration available that are the results of bug fixes and feature improvements.

KLEE [5] symbolic execution tool also used these coreutils in order to evaluate their tool. We build our incremental approach above the codebase of KLEE, these utilites were found very helpful in comparing our results with the original KLEE execution results.

We ran KLEE on all Coreutils for one hour and chose the 85 utilities for which KLEE covered maximum paths in this time. We chose version 7.1 as the base version of the Coreutils and then incrementally test one minor update release that is version 7.2 and one major update release that is version 8.1. We ran KLEE on base version completely and then incrementally ran version 7.2 and version 8.1.

[1] http://www.gnu.org/s/Coreutils

## 5.1 Methodology

We carried out our incremental experiments on the Texas Advanced Computing Center (TACC)[2]. Their Lonestar Linux cluster ensures unfailing running of these experimentations because they only allow one job to the processes while running tests.

We first generated complete test suite for base version 7.1 on specific depth taken from our depth analysis which we executed earlier. We performed incremental symbolic analysis on version 7.2 providing the test suite we gathered in base version. Then we executed version 8.1 incrementally by giving version 7.2 test suite as an input. For the sake of comparison we also executed the unmodified symbolic executor on version 7.2 and version 8.1 to generate complete test suites.

Every utility in the Coreutils are executed independently on different machine in TACC system but each execution of the utility is checked on the same machine with and without incremental technique.

## 5.2 Experimental Results

Table 5.1 and Table 5.2 show the results of all 85 programs we tested. First column in every version has time (in seconds) taken for the whole process to execute symbolically, second column tells the time took symbolic execution to solve each utility path conditions by the SMT solver, third column shows the number of queries sent to the solver and last column shows the number of test cases produced by the KLEE for each version. In most of the utilities number of test cases are same in all versions of the Coreutil programs. But the execution time in symbolic analysis with and without incremental approach is significantly

[2] http://tacc.utexas.edu

different. Incremental symbolic execution is almost 73% faster checking the same utility for the same number of test cases.

If we summarize the results on the basis of time required testing complete Coreutils programs, our incremental approach in about 73% faster and we are saving about 80% solver time by giving 67% less queries. Figure 5.1 shows a plot of the execution time of all 85 utilities with and without using our incremental approach on version 7.2 (in incremental seeds of version 7.1 were used). Similarly in Figure 5.2 execution time plot for symbolic execution of major Coreutils version 8.2 is shown without incremental testing and with incremental by providing test cases generated in version 7.2. Time taken for each utility with incremental approach is shown as squares in the plot while time taken without using incremental approach is shown as circles.

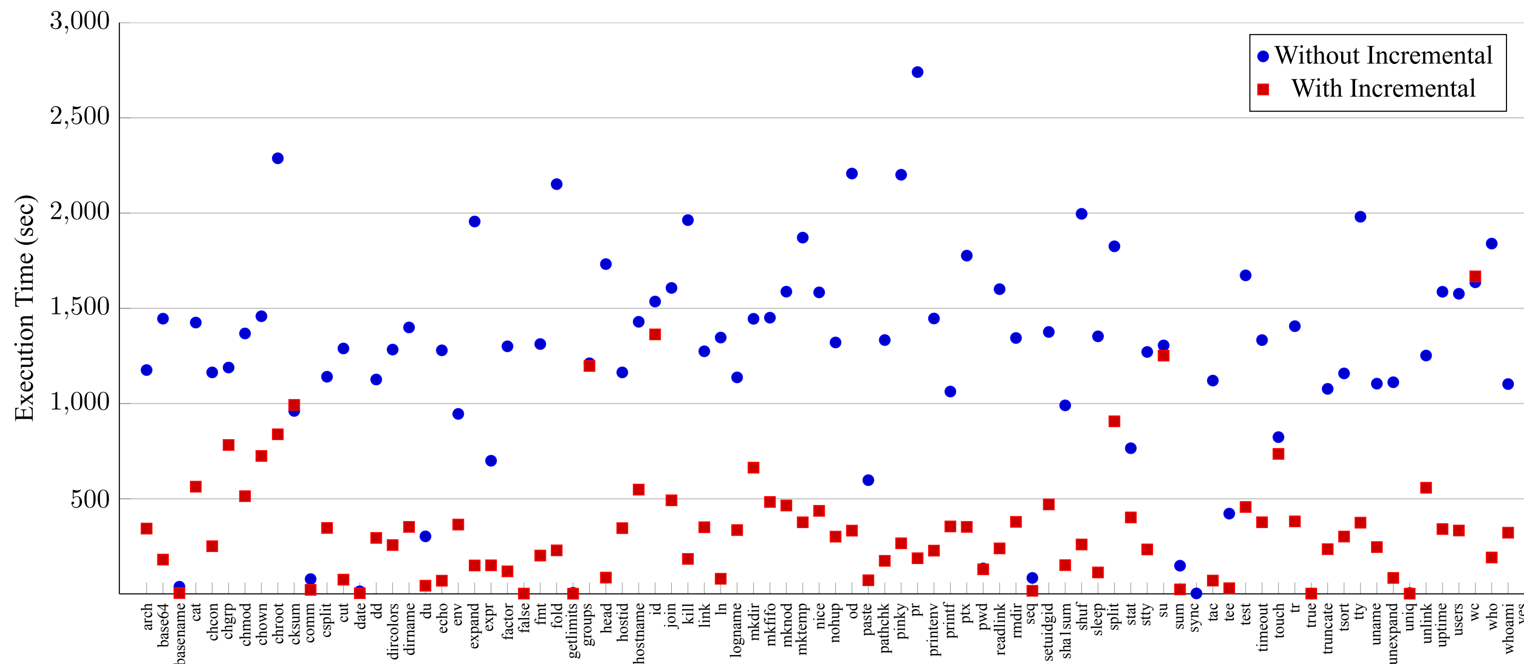


Figure 5.1: Execution time of 85 program from GNU Coreutils suite version 7.2 using incremental and non-incremental approach. Square boxes show the time for symbolic execution time by using incremental testing providing seeds from version 7.1. Circle shows the execution time of version 7.2 without using incremental technique.

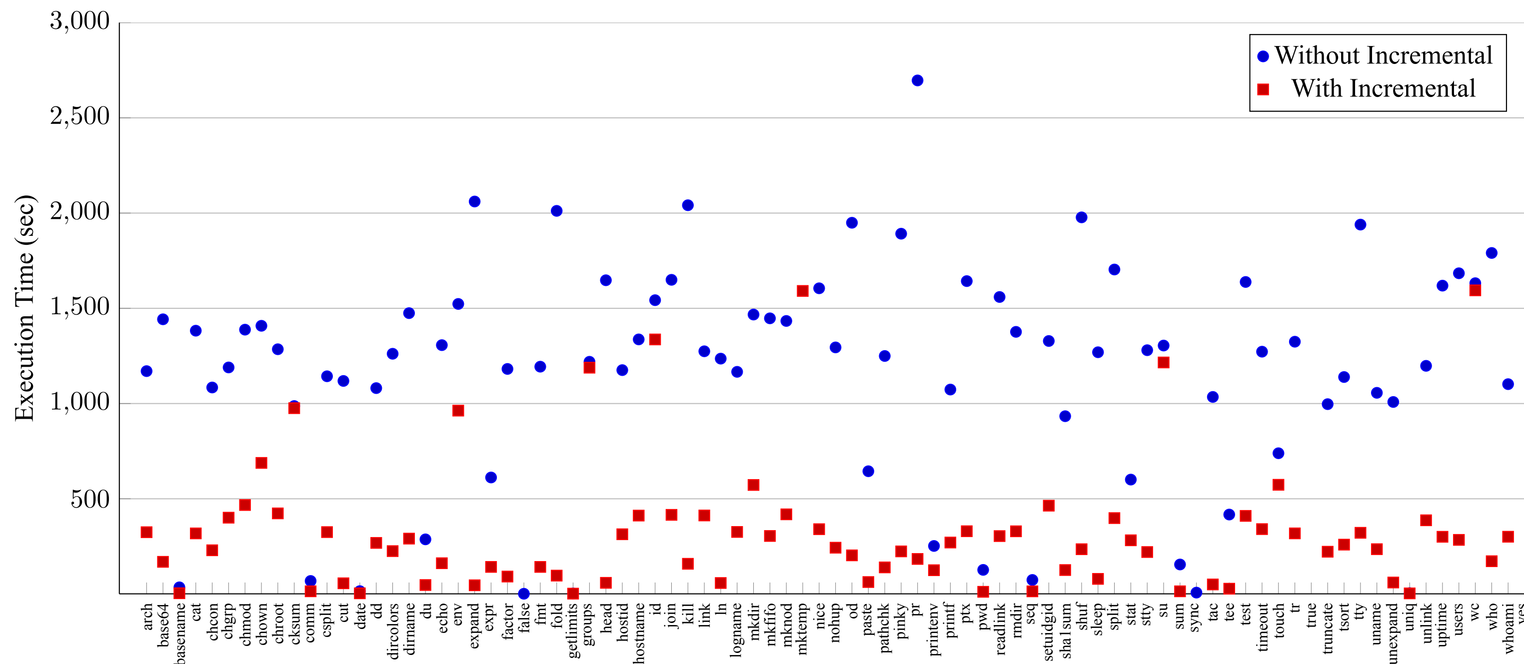


Figure 5.2: Execution time of 85 program from GNU Coreutils suite version 8.1 using incremental and non-incremental approach. Square boxes show the time for symbolic execution time by using incremental testing providing seeds from version 7.2. Circle shows the execution time of version 8.1 without using incremental technique.

## 5.3 Discussion on results

In symbolic execution, major portion of time is consumed at the solver side where it finds the solution of the path condition. In our approach we are reducing the number of queries sent to the solver. Finding a solution of a path condition using any solver is very costly. Our approach just compares the new path condition with the old path conditions and on successful comparison we just add those tests to the new version's test suite. Normally in new version of the program most of the source code remains unchanged if it is not very major update. That is why large number of test cases are reused in the process of building new test suite. Only few test cases were solved for concrete inputs using solver. These calls are very less in the incremental testing, that is shown in our analysis as well in Table 5.1. This small number of calls to the solver is actual cause of time saving in the incremental testing.

There are some utilities in the evaluation where numbers of test cases generated are different in the incremental and non-incremental version. That situation arises when those utilities call some external function calls to the operating system and those calls results in different manner. This happens because KLEE environment system is incomplete and did not entertain some system calls. So we checked those utilities independently and found out that they are using syscalls in certain areas that are not being executed symbolically during the analysis but their results are used in the program analysis. So these results turn out to be different in incremental and non-incremental. In case of different results from the syscall, the path condition varies from the ones we already have in the test suite and fails to validate. KLEE behave differently on these utilities each time when we executed them because of inbuilt limitations of the KLEE. That is the reason we have different number of test cases in our incremental and non-incremental version of the program.

Table 5.1: Symbolic execution results for 85 program from GNU Coreutils suite of Unix utilities of version 7.1, 7.2 (with and without incremental)

| **CoreUtil** | v7.1 | | | | v7.2 wo/ Incremental | | | | v7.2 w/ Incremental 7.1 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Time (seconds) | Solver | Tests | Queries | Time (seconds) | Solver | Tests | Queries | Tests (seconds) | Solver | Tests | Queries |
| arch | 1249 | 1030 | 3896 | 5106 | 1176 | 991 | 3896 | 5108 | 344 | 270 | 3896 | 1169 |
| base64 | 1485 | 1401 | 1516 | 3011 | 1446 | 1386 | 1516 | 3010 | 181 | 158 | 1516 | 1419 |
| basename | 45 | 22 | 377 | 643 | 38 | 20 | 377 | 643 | 5 | 0 | 377 | 150 |
| cat | 972 | 768 | 3358 | 4719 | 1425 | 1152 | 4212 | 5136 | 564 | 355 | 4212 | 1900 |
| chcon | 1206 | 984 | 3876 | 5123 | 1164 | 981 | 3876 | 5123 | 251 | 177 | 3876 | 1209 |
| chgrp | 1311 | 929 | 3584 | 5177 | 1189 | 879 | 3584 | 5177 | 782 | 190 | 3584 | 1557 |
| chmod | 1540 | 1315 | 4011 | 6782 | 1368 | 1221 | 4011 | 6782 | 514 | 438 | 4011 | 2713 |
| chown | 1626 | 1152 | 3594 | 5886 | 1458 | 1056 | 3594 | 5886 | 725 | 359 | 3594 | 2259 |
| chroot | 1424 | 1182 | 3885 | 4709 | 2288 | 2034 | 5714 | 6661 | 839 | 658 | 5714 | 2747 |
| cksum | 1016 | 1010 | 88 | 348 | 961 | 958 | 88 | 348 | 993 | 991 | 88 | 115 |
| comm | 81 | 44 | 611 | 1466 | 78 | 42 | 611 | 1466 | 22 | 2 | 611 | 803 |
| csplit | 1271 | 1061 | 3754 | 5739 | 1141 | 987 | 3721 | 5664 | 347 | 273 | 3754 | 1846 |
| cut | 1271 | 1039 | 4580 | 5349 | 1289 | 1058 | 4580 | 5349 | 76 | 21 | 4580 | 716 |
| date | 21 | 8 | 146 | 538 | 14 | 7 | 146 | 538 | 4 | 0 | 146 | 300 |
| dd | 1224 | 1016 | 4053 | 7622 | 1126 | 954 | 4053 | 7608 | 295 | 223 | 4053 | 3506 |
| dircolors | 1424 | 1194 | 4208 | 5330 | 1283 | 1097 | 4208 | 5331 | 257 | 176 | 4208 | 1090 |
| dirname | 1591 | 1338 | 3885 | 4618 | 1400 | 1233 | 3885 | 4617 | 352 | 276 | 3885 | 652 |
| du | 336 | 223 | 1882 | 2315 | 303 | 216 | 1882 | 2315 | 44 | 1 | 1882 | 320 |
| echo | 1378 | 1159 | 4495 | 4863 | 1279 | 1089 | 4495 | 4862 | 70 | 19 | 4072 | 616 |
| env | 1081 | 705 | 3422 | 3799 | 946 | 616 | 3422 | 3799 | 365 | 11 | 3422 | 321 |
| expand | 2326 | 1992 | 5916 | 6875 | 1956 | 1743 | 5920 | 6881 | 150 | 45 | 6493 | 1634 |
| expr | 766 | 644 | 1793 | 6264 | 700 | 593 | 1793 | 6260 | 151 | 113 | 1793 | 2334 |
| factor | 1415 | 1151 | 5016 | 6920 | 1301 | 1053 | 5016 | 6919 | 119 | 31 | 5016 | 1921 |

| CoreUtil | v7.1 | | | | v7.2 wo/ Incremental | | | | v7.2 w/ Incremental 7.1 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Time (seconds) | Solver | Tests | Queries | Time (seconds) | Solver | Tests | Queries | Tests (seconds) | Solver | Tests | Queries |
| false | 2 | 1 | 9 | 174 | 1 | 0 | 9 | 174 | 2 | 0 | 9 | 31 |
| fmt | 1417 | 1154 | 5368 | 6471 | 1313 | 1080 | 5533 | 6628 | 202 | 113 | 5533 | 1265 |
| fold | 2486 | 2080 | 7754 | 9310 | 2152 | 1898 | 7754 | 9310 | 229 | 20 | 7754 | 1623 |
| getlimits | 6 | 1 | 20 | 192 | 6 | 1 | 20 | 192 | 2 | 0 | 20 | 79 |
| groups | 1279 | 67 | 666 | 1030 | 1212 | 65 | 666 | 1030 | 1197 | 26 | 666 | 236 |
| head | 1930 | 1639 | 5120 | 6130 | 1732 | 1530 | 5120 | 6125 | 86 | 17 | 5120 | 1000 |
| hostid | 1243 | 1021 | 3860 | 5064 | 1164 | 977 | 3860 | 5064 | 346 | 264 | 3860 | 1165 |
| hostname | 1446 | 1200 | 4430 | 5751 | 1429 | 1212 | 4430 | 5751 | 548 | 366 | 4430 | 1291 |
| id | 1626 | 304 | 1607 | 2510 | 1536 | 288 | 1607 | 2510 | 1363 | 114 | 1607 | 841 |
| join | 1713 | 1456 | 4950 | 8295 | 1607 | 1431 | 4950 | 8296 | 492 | 409 | 4950 | 3286 |
| kill | 2135 | 1850 | 4254 | 5422 | 1963 | 1791 | 4361 | 5579 | 184 | 101 | 4254 | 1757 |
| link | 1374 | 1139 | 4079 | 6286 | 1274 | 1088 | 4079 | 6287 | 351 | 270 | 4079 | 2179 |
| ln | 1414 | 1197 | 4107 | 7890 | 1346 | 1145 | 4107 | 7891 | 80 | 11 | 4107 | 3726 |
| logname | 1233 | 1015 | 3860 | 5063 | 1137 | 957 | 3860 | 5064 | 336 | 259 | 3860 | 1162 |
| mkdir | 1593 | 1359 | 4227 | 6621 | 1445 | 1264 | 4227 | 6621 | 664 | 580 | 4227 | 2351 |
| mkfifo | 1571 | 1315 | 4550 | 5904 | 1451 | 1269 | 4550 | 5905 | 483 | 294 | 4550 | 1325 |
| mknod | 1596 | 1328 | 4828 | 6328 | 1588 | 1403 | 4828 | 6335 | 465 | 272 | 4828 | 1470 |
| mktemp | 2144 | 1816 | 5768 | 7870 | 1871 | 1653 | 5768 | 7870 | 377 | 284 | 5768 | 2069 |
| nice | 1939 | 1656 | 5137 | 8773 | 1584 | 1407 | 5078 | 8699 | 437 | 286 | 5137 | 3594 |
| nohup | 1484 | 1184 | 5550 | 7182 | 1321 | 1093 | 5550 | 7182 | 301 | 192 | 5550 | 1605 |
| od | 2438 | 2057 | 6956 | 7963 | 2208 | 1974 | 6959 | 7968 | 333 | 260 | 6956 | 3480 |
| paste | 631 | 493 | 2436 | 6438 | 598 | 494 | 2426 | 6828 | 72 | 27 | 2426 | 3511 |
| pathchk | 1381 | 1143 | 4768 | 5891 | 1333 | 1123 | 4768 | 5891 | 174 | 98 | 4768 | 1060 |
| pinky | 2418 | 1914 | 5632 | 5925 | 2201 | 1851 | 5632 | 5925 | 266 | 5 | 5632 | 277 |
| pr | 2893 | 2863 | 484 | 1027 | 2740 | 2722 | 484 | 1027 | 188 | 179 | 484 | 391 |
| printenv | 1723 | 1456 | 4501 | 4840 | 1447 | 1272 | 4501 | 4840 | 228 | 43 | 4501 | 314 |
| printf | 1236 | 1069 | 2709 | 14166 | 1063 | 933 | 2487 | 13977 | 355 | 276 | 2522 | 9912 |

| CoreUtil | v7.1 | | | | v7.2 wo/ Incremental | | | | v7.2 w/ Incremental 7.1 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Time (seconds) | Solver | Tests | Queries | Time (seconds) | Solver | Tests | Queries | Tests (seconds) | Solver | Tests | Queries |
| ptx | 2085 | 1836 | 3507 | 6893 | 1777 | 1645 | 3511 | 6917 | 352 | 293 | 3505 | 3363 |
| pwd | 8 | 3 | 35 | 212 | 134 | 86 | 1037 | 1244 | 130 | 83 | 1037 | 1162 |
| readlink | 1730 | 1474 | 4444 | 5256 | 1601 | 1429 | 4444 | 5256 | 240 | 167 | 4444 | 709 |
| rmdir | 1542 | 1284 | 4776 | 6406 | 1344 | 1160 | 4776 | 6405 | 379 | 183 | 4776 | 1576 |
| seq | 89 | 61 | 390 | 678 | 85 | 57 | 390 | 678 | 17 | 3 | 390 | 138 |
| setuidgid | 1486 | 1143 | 2792 | 13288 | 1376 | 1075 | 2792 | 13288 | 471 | 248 | 2792 | 10442 |
| sha1sum | 1052 | 868 | 3796 | 4152 | 991 | 826 | 3796 | 4151 | 152 | 89 | 3796 | 341 |
| shuf | 2223 | 1927 | 5789 | 8425 | 1996 | 1797 | 5789 | 8425 | 260 | 193 | 5789 | 2552 |
| sleep | 1535 | 1253 | 5363 | 6058 | 1353 | 1122 | 5363 | 6058 | 113 | 19 | 5363 | 672 |
| split | 1937 | 1636 | 5174 | 7507 | 1826 | 1553 | 4924 | 6221 | 907 | 671 | 4927 | 2814 |
| stat | 796 | 300 | 2177 | 2754 | 766 | 292 | 2177 | 2753 | 402 | 2 | 2177 | 486 |
| stty | 1331 | 1114 | 3998 | 9440 | 1271 | 1098 | 3998 | 9440 | 234 | 165 | 3998 | 5050 |
| su | 1358 | 97 | 947 | 1378 | 1305 | 93 | 947 | 1378 | 1252 | 27 | 947 | 390 |
| sum | 166 | 108 | 1128 | 1373 | 149 | 104 | 1128 | 1373 | 24 | 1 | 1128 | 162 |
| sync | 6 | 2 | 35 | 212 | 3 | 1 | 35 | 212 | 1 | 0 | 35 | 41 |
| tac | 1169 | 964 | 3960 | 4812 | 1121 | 923 | 3961 | 4813 | 71 | 7 | 3961 | 665 |
| tee | 460 | 354 | 2126 | 2411 | 422 | 339 | 2126 | 2411 | 31 | 1 | 2126 | 161 |
| test | 1894 | 1597 | 4959 | 6455 | 1673 | 1482 | 4959 | 6456 | 457 | 371 | 4959 | 1299 |
| timeout | 1484 | 1250 | 3982 | 6561 | 1333 | 1164 | 3982 | 6561 | 377 | 302 | 3982 | 2535 |
| touch | 975 | 879 | 1703 | 8748 | 824 | 754 | 1703 | 7956 | 736 | 602 | 1703 | 8396 |
| tr | 1539 | 1287 | 4631 | 6094 | 1406 | 1227 | 4628 | 6088 | 381 | 298 | 4629 | 1565 |
| true | 2 | 1 | 9 | 174 | 2 | 0 | 9 | 174 | 2 | 0 | 9 | 31 |
| truncate | 1126 | 966 | 3141 | 4766 | 1077 | 933 | 3141 | 4766 | 236 | 190 | 3141 | 1475 |
| tsort | 1268 | 1063 | 3498 | 5237 | 1158 | 1003 | 3498 | 5237 | 302 | 236 | 3498 | 1708 |
| tty | 2170 | 1819 | 5720 | 7115 | 1981 | 1758 | 5720 | 7115 | 375 | 262 | 5720 | 1357 |
| uname | 1175 | 967 | 3778 | 5060 | 1104 | 927 | 3778 | 5060 | 246 | 177 | 3778 | 1251 |
| unexpand | 1124 | 905 | 4141 | 4699 | 1112 | 896 | 4159 | 4721 | 85 | 9 | 4141 | 615 |

| **CoreUtil** | v7.1 | | | | v7.2 wo/ Incremental | | | | v7.2 w/ Incremental 7.1 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Time (seconds) | Solver | Tests | Queries | Time (seconds) | Solver | Tests | Queries | Tests (seconds) | Solver | Tests | Queries |
| uniq | 10 | 3 | 59 | 250 | 5 | 2 | 59 | 250 | 2 | 0 | 59 | 59 |
| unlink | 1307 | 1088 | 3882 | 5284 | 1252 | 1096 | 3882 | 5285 | 558 | 378 | 3882 | 1348 |
| uptime | 1793 | 1514 | 4196 | 4984 | 1587 | 1403 | 4196 | 4986 | 341 | 255 | 4196 | 673 |
| users | 1751 | 1512 | 4196 | 4983 | 1576 | 1399 | 4196 | 4985 | 334 | 255 | 4196 | 673 |
| wc | 1678 | 1645 | 660 | 1047 | 1637 | 1613 | 660 | 1047 | 1668 | 1655 | 660 | 341 |
| who | 2065 | 1693 | 5199 | 5563 | 1840 | 1546 | 5199 | 5563 | 192 | 4 | 5199 | 302 |
| whoami | 1166 | 961 | 3860 | 5067 | 1102 | 930 | 3860 | 5064 | 322 | 245 | 3860 | 1164 |
| yes | 1 | 0 | 2 | 2 | 0 | 0 | 2 | 2 | 0 | 0 | 2 | 2 |

Table 5.2: Symbolic execution results for 85 program from GNU Coreutils suite of Unix utilities of version 8.1 (with and without incremental)

| **CoreUtil** | v8.1 wo/ Incremental | | | | v8.1 w/ Incremental 7.2 | | | |
|---|---|---|---|---|---|---|---|---|
| | Time (seconds) | Solver | Test | Queries | Tests (seconds) | Solver | Tests | Queries |
| arch | 1170 | 1025 | 3896 | 5101 | 325 | 275 | 3896 | 1163 |
| base64 | 1442 | 1376 | 1516 | 3009 | 169 | 146 | 1516 | 1378 |
| basename | 35 | 19 | 377 | 643 | 4 | 0 | 377 | 101 |
| cat | 1383 | 1147 | 4212 | 5136 | 318 | 136 | 4212 | 827 |
| chcon | 1084 | 945 | 3876 | 5099 | 230 | 179 | 3876 | 1193 |
| chgrp | 1190 | 886 | 3584 | 5134 | 401 | 188 | 3584 | 1512 |
| chmod | 1388 | 1242 | 4011 | 6747 | 467 | 432 | 4011 | 2668 |
| chown | 1408 | 999 | 3594 | 5841 | 688 | 364 | 3594 | 2203 |
| chroot | 1285 | 1119 | 3982 | 4815 | 423 | 354 | 5811 | 1032 |
| cksum | 987 | 984 | 88 | 349 | 976 | 974 | 88 | 115 |
| comm | 69 | 41 | 611 | 1464 | 14 | 1 | 611 | 747 |
| csplit | 1143 | 1008 | 3754 | 5730 | 325 | 277 | 3754 | 1824 |
| cut | 1119 | 962 | 4580 | 5349 | 56 | 20 | 4580 | 606 |
| date | 15 | 7 | 146 | 536 | 4 | 0 | 146 | 248 |
| dd | 1081 | 935 | 4053 | 7582 | 268 | 209 | 4053 | 3429 |
| dircolors | 1261 | 1116 | 4208 | 5327 | 225 | 178 | 4208 | 1012 |
| dirname | 1475 | 1324 | 3885 | 4615 | 291 | 255 | 3885 | 643 |
| du | 287 | 216 | 1900 | 2333 | 47 | 1 | 1900 | 326 |
| echo | 1307 | 1163 | 4729 | 5098 | 162 | 91 | 4729 | 1235 |
| env | 1523 | 1317 | 4321 | 5744 | 963 | 748 | 4321 | 3802 |
| expand | 2061 | 1863 | 5916 | 6442 | 46 | 6 | 5919 | 505 |
| expr | 612 | 515 | 1793 | 6264 | 142 | 113 | 1793 | 2300 |
| factor | 1182 | 997 | 5016 | 6918 | 92 | 29 | 5016 | 1811 |

| CoreUtil | v8.1 wo/ Incremental | | | | v8.1 w/ Incremental 7.2 | | | |
|---|---|---|---|---|---|---|---|---|
| | Time (seconds) | Solver | Test | Queries | Tests (seconds) | Solver | Tests | Queries |
| false | 1 | 1 | 9 | 174 | 0 | 0 | 9 | 31 |
| fmt | 1194 | 1025 | 5368 | 6470 | 142 | 91 | 5368 | 1087 |
| fold | 2012 | 1758 | 7754 | 9308 | 97 | 14 | 7754 | 1472 |
| getlimits | 3 | 1 | 20 | 192 | 2 | 0 | 20 | 36 |
| groups | 1219 | 63 | 666 | 1030 | 1188 | 26 | 666 | 236 |
| head | 1647 | 1477 | 5120 | 6128 | 59 | 15 | 5120 | 896 |
| hostid | 1176 | 1031 | 3860 | 5062 | 314 | 262 | 3860 | 1153 |
| hostname | 1337 | 1169 | 4430 | 5736 | 412 | 369 | 4430 | 1264 |
| id | 1543 | 288 | 1607 | 2508 | 1337 | 114 | 1607 | 831 |
| join | 1650 | 1472 | 4950 | 8277 | 416 | 373 | 4950 | 3208 |
| kill | 2041 | 1874 | 4361 | 5579 | 159 | 98 | 4254 | 1471 |
| link | 1274 | 1126 | 4079 | 6265 | 412 | 273 | 4079 | 2145 |
| ln | 1236 | 1098 | 4108 | 7891 | 58 | 10 | 4108 | 3709 |
| logname | 1166 | 1022 | 3860 | 5059 | 326 | 275 | 3860 | 1167 |
| mkdir | 1467 | 1273 | 4227 | 6589 | 573 | 530 | 4227 | 2310 |
| mkfifo | 1447 | 1279 | 4550 | 5893 | 305 | 268 | 4550 | 1315 |
| mknod | 1434 | 1205 | 4828 | 6311 | 418 | 270 | 4828 | 1441 |
| mktemp | 4224 | 3996 | 5846 | 28266 | 1591 | 1507 | 5846 | 22042 |
| nice | 1605 | 1413 | 5078 | 8674 | 340 | 285 | 5137 | 3461 |
| nohup | 1295 | 1100 | 5550 | 7138 | 243 | 187 | 5550 | 1550 |
| od | 1949 | 1716 | 6959 | 7972 | 203 | 145 | 6959 | 2488 |
| paste | 645 | 501 | 2472 | 6333 | 63 | 25 | 2426 | 3879 |
| pathchk | 1250 | 1098 | 4768 | 5853 | 140 | 93 | 4768 | 989 |
| pinky | 1892 | 1547 | 5632 | 5925 | 224 | 4 | 5632 | 204 |
| pr | 2696 | 2678 | 484 | 1027 | 184 | 179 | 484 | 345 |
| printenv | 252 | 199 | 1627 | 1962 | 125 | 75 | 1627 | 1074 |
| printf | 1073 | 970 | 2594 | 14155 | 270 | 224 | 2522 | 9280 |

| CoreUtil | v8.1 wo/ Incremental | | | | v8.1 w/ Incremental 7.2 | | | |
|---|---|---|---|---|---|---|---|---|
| | Time (seconds) | Solver | Test | Queries | Tests (seconds) | Solver | Tests | Queries |
| ptx | 1643 | 1513 | 3508 | 6900 | 330 | 286 | 3508 | 3360 |
| pwd | 127 | 83 | 1037 | 1244 | 12 | 0 | 1037 | 111 |
| readlink | 1559 | 1402 | 4447 | 5254 | 304 | 169 | 4447 | 708 |
| rmdir | 1376 | 1210 | 4776 | 6374 | 329 | 187 | 4776 | 1539 |
| seq | 74 | 55 | 390 | 678 | 14 | 3 | 390 | 182 |
| setuidgid | 1328 | 1049 | 2792 | 13282 | 464 | 249 | 2792 | 10332 |
| sha1sum | 934 | 799 | 3796 | 4152 | 126 | 87 | 3796 | 311 |
| shuf | 1978 | 1741 | 5789 | 8420 | 235 | 191 | 5789 | 2482 |
| sleep | 1269 | 1084 | 5363 | 6058 | 80 | 17 | 5363 | 645 |
| split | 1704 | 1433 | 4924 | 6221 | 399 | 243 | 4924 | 1204 |
| stat | 601 | 269 | 2129 | 2852 | 282 | 2 | 2124 | 623 |
| stty | 1280 | 1116 | 3998 | 9360 | 220 | 165 | 3998 | 4916 |
| su | 1305 | 91 | 947 | 1378 | 1216 | 27 | 947 | 341 |
| sum | 155 | 106 | 1128 | 1373 | 14 | 1 | 1128 | 111 |
| sync | 7 | 2 | 35 | 212 | 1 | 0 | 35 | 41 |
| tac | 1035 | 892 | 3961 | 4815 | 50 | 6 | 3960 | 556 |
| tee | 417 | 331 | 2126 | 2411 | 28 | 1 | 2126 | 111 |
| test | 1638 | 1449 | 4959 | 6452 | 410 | 349 | 4959 | 1272 |
| timeout | 1272 | 1125 | 3982 | 6553 | 341 | 296 | 3982 | 2470 |
| touch | 739 | 687 | 1448 | 7752 | 573 | 551 | 1439 | 8231 |
| tr | 1325 | 1166 | 4629 | 6078 | 318 | 273 | 4628 | 1590 |
| true | 1 | 0 | 9 | 174 | 0 | 0 | 9 | 31 |
| truncate | 997 | 863 | 3142 | 4781 | 222 | 186 | 3142 | 1542 |
| tsort | 1139 | 1010 | 3498 | 5235 | 259 | 218 | 3498 | 1656 |
| tty | 1939 | 1666 | 5720 | 7098 | 321 | 260 | 5720 | 1329 |
| uname | 1056 | 923 | 3778 | 5037 | 235 | 183 | 3778 | 1215 |
| unexpand | 1008 | 857 | 4159 | 4706 | 60 | 7 | 4141 | 539 |

| **CoreUtil** | v8.1 wo/ Incremental | | | | v8.1 w/ Incremental 7.2 | | | |
|---|---|---|---|---|---|---|---|---|
| | Time (seconds) | Solver | Test | Queries | Tests (seconds) | Solver | Tests | Queries |
| uniq | 5 | 2 | 59 | 250 | 3 | 0 | 59 | 59 |
| unlink | 1198 | 1055 | 3882 | 5277 | 387 | 349 | 3882 | 1330 |
| uptime | 1619 | 1422 | 4196 | 4984 | 301 | 253 | 4196 | 676 |
| users | 1684 | 1515 | 4196 | 4982 | 284 | 249 | 4196 | 662 |
| wc | 1632 | 1608 | 660 | 1048 | 1594 | 1586 | 660 | 301 |
| who | 1791 | 1504 | 5199 | 5563 | 172 | 3 | 5199 | 243 |
| whoami | 1102 | 956 | 3860 | 5059 | 301 | 247 | 3860 | 1153 |
| yes | 0 | 0 | 2 | 2 | 0 | 0 | 2 | 2 |

# Chapter 6

# RELATED WORK

In the language of programming, imperative programs are constituted of those commands that change the state of the programs. The symbolic execution of imperative program was introduced by Clarke [9] and King [16]. The symbolic execution has now progressed from these traditional methods in the past decade. Now object oriented symbolic execution and *lazy initialization* are being used for pointer aliasing in Generalized symbolic execution [15] and PREfixx [4] has proven effectively good in finding bug on real code executed by symbolic execution.

Owing to the effectiveness of symbolic execution, extensive studies are being done from the past 7 years. DART [13] has experimented with symbolic execution by combining it with concrete execution with the result that it would get the branch conditions apart from execution path. In order to avoid the condition in which function executes on another path, DART negates the last branch condition for the purpose of making a new path condition. And hence, DART emphasize on those path conditions that are comprised of integers. This barrier was overcome through SMART [2] that came up with inter-procedural static analysis techniques to calculate the summaries of procedures. SMART [12] also comes in handy as it reduce the paths that DART explores. As far as CUTE [21] is concerned it helps in handling constraints on references by extending to DART.

The technique of negation of branch predicates is also utilized by EGT [6] and EXE [7].

These also make use of symbolic execution in order to generate test cases. The purpose of these is to enhance the accuracy of symbolic pointer analysis to control pointer arithmetic and bit-level memory locations. In this regard the most recent tool till date in the KLEE [5] when it comes to the EGT/EXE family. In the realm of industry and academics KLEE has been extensively used by many users for being open sourced. KLEE works on those programs written in C/C++ that are unmodified as well as on LLVM byte code [1]. KLEE has proven effective for many off the shelf programs. KLEE is also being utilized as enabling technology by ranged symbolic execution.

## 6.1 Incremental Symbolic Execution based on Static Analysis

In order to optimize the symbolic execution of affected paths, the directed incremental symbolic execution [19] is used to tell apart the differences in program versions. The purpose of this is to avoid the modified behavior in the affected paths. In those program versions in which previously symbolic execution has been done, the motivation of this procedure is to avoid symbolically executing paths in them. For this purpose, in order to single out the affected locations a reachability analysis is utilized that then guide the symbolic exploration.

DISE [19] makes use of static analysis to detect code blocks that are caused by a change. After detecting code blocks DISE makes use of dynamic analysis to modify the execution tree for symbolic execution. The execution of DISE takes place only after static analysis of both CFGs programs so it generate only affected path conditions. Usually the user requires a test suite with the use of DISE to check which conditions get outdated but it is not clear as to how it is done. The advantage that our technique holds is that it operates on dynamic analysis alone. With the help of previous execution, the range of symbolic execution is determined through validating path conditions. The advantage of a pure dynamic technique is its ability to estimate precisely as compared to DISE that over-estimate the influence of

a change that occurs. The benefits of DISE and dynamic analysis can be done if in an extension, ideas from DISE are combined with dynamic analysis in order for the static analysis to improve a portion and in this way more benefits can be attained.

Katch [18] combines static and dynamic analysis for increased coverage of software patches. Katch uses heuristics basic on static analysis to select tests from a manual test suite that are most likely to hit modified code and then it uses heuristics based on dynamic analysis to change the test inputs to increase chances of hitting modified code. The dynamic analysis is built upon symbolic execution. Since the problem of code reachability is undecidable in general, Katch hopes that the heuristics will lead to modified code in most cases. Our idea of incremental symbolic execution does not depend a manual test suite and instead does bounded exhaustive symbolic execution. Our claim is that if the fault can be found using standard symbolic execution, we will be able to find it in less time. Or, in other words, our incremental symbolic execution can explore the program up to a deeper depth bound in the same time due to the time saved by incremental execution.

## 6.2 Incremental Symbolic Execution based on caching solutions to path constraints

Green [28] is a technique of caching that store path conditions and then store their solution. The benefit of Green is that it slices the path condition so that these can be reduced and hence analyzed so satisfy-ability. Once done, heuristic is performed on them that is based on canonization in order to maximize the chance of getting matches with the path and finally stores them. Green provides an effective checking in incremental situations although it is not an incremental symbolic execution tool. The benefit in using Green for the incremental setting is that many path conditions may be matched from the previous execution and hence solving is avoided. Path conditions are complicated structures which involves some work

done in order to compare them and map the variations even when Green is used. The process of comparison becomes more complicated with the increase in the size of cache. In comparison the process of validation of a path condition is much quicker and there is no need of any lookup as well.

Memoized symbolic execution [29] is an innovative technique that uses the results of previously run symbolic execution and then stores these in a trie-based data structure. Memoized uses these again for the maintenance and updation of trie during the next run of symbolic execution on the modified program. The results in technique based on ranged symbolic execution is efficient as compared to the incremental symbolic execution for being cost effective concrete test versus storing and the comparison of symbolic execution results in tries.

## 6.3 Other ways to scale symbolic execution

Some recent studies has put forward some technique for parallel symbolic execution [23, 27, 24]. ParSym [23] used this technique of parallel symbolic execution while taking every path exploration as a unit of work and to distribute work among parallel workers it made use of a central server. Although this technique is beneficial in applying a direct approach for parallelization [14, 7] yet symbolic constraints for every branch needs to be communicated among the workers. On the other hand, static partitioning [27] makes use of symbolic execution's first shallow run in order to minimize the communication overhead during the process of running parallel symbolic execution. For this purpose, pre-conditions are made that are made by using conjunctions of clauses on path conditions that come in the process of shallow run and to restrict the symbolic execution by each worker. This is done to program such paths that satisfy the pre-condition for that worker’s path exploration. The drawback of creating the pre-conditions is that the ranges of the workers who explore overlap, because of

which the efforts go wasted. Ranged symbolic execution [24] removes this very discrepancy and utilizes the dynamic load balancing so that workers may not encounter nay overlap and hence the communication is kept low. Eventually the incremental analysis can have a substantial influence on the scalability.

Hybrid concolic testing [17] guides the symbolic execution periodically by searching randomly. This is helpful in increasing the code coverage. The flaw that this technique holds is that it explores overlapping range as well while hopping from symbolic execution of one area of code to the other. Compositional symbolic execution [12] avoids re-exploration by making use of method summaries. Staged symbolic execution [25] is applied on symbolic execution in stages instead of traditional approaches of it implementation. In fact the incremental symbolic execution is staging when generalized, stages are the code version and codes infer codes automatically.

All the above mentioned techniques can be utilized in combination with incremental execution.

# Chapter 7

# CONCLUSION

In this work, we showed how to scale symbolic execution for efficiently analyzing program increments. The basis innovation of our work is to apply symbolic execution in increments with only dynamic analysis. Our approach does not use any form of static analysis or caching of previous results for completely automated incremental testing.

The key idea of our technique is that we can use previously generated test cases for identification of code changes in the new version. Checking constraints is easy and time saving job instead of solving them.As an empowering technology we implemented our technique over the open-source tool KLEE that is state-of-the-art tool to execute C programs symbolically. Our results consist of testing 85 programs from Coreutils suite shows that our approach provides a significant speedup on increments testing. We finished execution in 73% less time by giving 67% less queries to the solver for 85 programs.

Our technique for automated software testing lead us to the place where we can test increments of the program more efficiently and effectively. We hope that our results would act as a benchmark for researchers of this field.

# Bibliography

[1] Vikram Adve, Chris Lattner, Michael Brukman, Anand Shukla, and Brian Gaeke. LLVA: A Low-level Virtual Instruction Set Architecture. In *Proc. 36$^{th}$ International Symposium on Microarchitecture (MICRO)*, pages 205--216, 2003.

[2] Saswat Anand, Corina S. Păsăreanu, and Willem Visser. JPF-SE: a Symbolic Execution Extension to Java PathFinder. In *Proc. 13$^{th}$ International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 134--138, 2007.

[3] Clark Barrett and Cesare Tinelli. CVC3. In *Proc. 19$^{th}$ International Conference on Computer Aided Verification (CAV)*, pages 298--302, 2007.

[4] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A Static Analyzer for Finding Dynamic Programming Errors. *Software Practice Experience* , 30(7):775--802, June 2000.

[5] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. 8$^{th}$ Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209--224, 2008.

[6] Cristian Cadar and Dawson Engler. Execution Generated Test Cases: How to make systems code crash itself. In *Proc. International SPIN Workshop on Model Checking of Software* , pages 2--23, 2005.

[7] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically Generating Inputs of Death. In *Proc. 13$^{th}$ Conference on Computer and Communications Security (CCS)*, pages 322--335, 2006.

[8] Lori A. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Transactions on Software Engineering* , 2(3):215--222, May 1976.

[9] Lori A. Clarke. *Test Data Generation and Symbolic Execution of Programs as an aid to Program Validation.* PhD thesis, University of Colorado at Boulder, 1976.

[10] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337--340, 2008.

[11] Bassem Elkarablieh, Ivan Garcia, Yuk Lai Suen, and Sarfraz Khurshid. Assertion-based Repair of Complex Data Structures. In *Proc. 22$^{nd}$ International Conference on Automated Software Engineering (ASE)*, pages 64--73, 2007.

[12] Patrice Godefroid. Compositional Dynamic Test Generation. In *Proc. 34$^{th}$ Symposium on Principles of Programming Languages (POPL)*, pages 47--54, 2007.

[13] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *Proc. 2005 Conference on Programming Languages Design and Implementation (PLDI)*, pages 213--223, 2005.

[14] Patrice Godefroid, Michael Y. Levin, and David A Molnar. Automated Whitebox Fuzz Testing. In *Proc. Network and Distributed System Security Symposium (NDSS)*, 2008.

[15] Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. Generalized Symbolic Execution for Model Checking and Testing. In *Proc. 9$^{th}$ International Conference*

*on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 553--568, 2003.

[16] James C. King. Symbolic Execution and Program Testing. *Communications ACM*, 19(7):385--394, July 1976.

[17] Rupak Majumdar and Koushik Sen. Hybrid Concolic Testing. In *Proc. 29$^{th}$ International Conference on Software Engineering (ICSE)*, pages 416--426, 2007.

[18] Cristian Cadar Paul Dan Marinescu. KATCH: High-Coverage Testing of Software Patches. In *Proc. 12$^{th}$ joint meeting of the European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*, 2013.

[19] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. Directed Incremental Symbolic Execution. In *Proc. 2011 Conference on Programming Languages Design and Implementation (PLDI)*, pages 504--515, 2011.

[20] David A. Ramos and Dawson R. Engler. Practical, Low-Effort Equivalence Verification of Real Code. In *Proc. 23$^{rd}$ International Conference on Computer Aided Verification (CAV)*, pages 669--685, 2011.

[21] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proc. 5$^{th}$ joint meeting of the European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 263--272, 2005.

[22] Chiyoung Seo, Sam Malek, and Nenad Medvidovic. Component-Level Energy Consumption Estimation for Distributed Java-Based Software Systems. In *Proc. 11$^{th}$ International Symposium on Component-Based Software Engineering*, pages 97--113, 2008.

[23] Junaid Haroon Siddiqui and Sarfraz Khurshid. ParSym: Parallel Symbolic Execution. In *Proc. 2nd International Conference on Software Technology and Engineering (ICSTE)*, pages V1: 405--409, 2010.

[24] Junaid Haroon Siddiqui and Sarfraz Khurshid. Scaling Symbolic Execution using Ranged Analysis. In *Proc. 27th Annual Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2012.

[25] Junaid Haroon Siddiqui and Sarfraz Khurshid. Staged Symbolic Execution. In *Proc. 27th Symposium on Applied Computing (SAC): Software Verification and Testing Track (SVT)*, 2012.

[26] Niklas Sörensson and Niklas Een. An Extensible SAT-solver. In *Proc. 6th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 502--518, 2003.

[27] Matt Staats and Corina Păsăreanu. Parallel Symbolic Execution for Structural Test Generation. In *Proc. 19th International Symposium on Software Testing and Analysis (ISSTA)*, pages 183--194, 2010.

[28] Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. Green: Reducing, Reusing and Recycling Constraints in Program Analysis. In *Proc. 2012joint meeting of the European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*, 2012.

[29] Guowei Yang, Corina S. Păsăreanu, and Sarfraz Khurshid. Memoized Symbolic Execution. In *Proc. 2012 International Symposium on Software Testing and Analysis (ISSTA)*, ISSTA 2012, pages 144--154, New York, NY, USA, 2012. ACM.

# Vita

Sarmad Makhdoom was born in Hafizabad, Pakistan on 30 June 1987, the son of Dr. Makhdoom Iqbal Qureshi. He received the Bachelor of Science degree in Computer Science from the Comsats Institute of Information Technology, Lahore, Pakistan with Campus Gold and Chancellor Gold medals, he applied to the Lahore Univeristy of Management and Sciences in Computer Science program. He was accepted and started graduate studies in August, 2012.

This dissertation was typeset with LaTeX[†] by the author.

[†] LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.