Air Force Institute of Technology
AFIT Scholar

Theses and Dissertations Student Graduate Works

9-14-2017

A Tree Locality-Sensitive Hash for Secure Software Testing

Camdon J. Cady

A TREE LOCALITY-SENSITIVE HASH FOR SECURE SOFTWARE TESTING

DISSERTATION

Camdon R Cady, Capt, USAF

AFIT-ENG-DS-17-S-005

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

The views expressed in this document are those of the author and do not reflect the official policy or position of the United States Air Force, the United States Department of Defense or the United States Government. This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

AFIT-ENG-DS-17-S-005

A TREE LOCALITY-SENSITIVE HASH FOR SECURE SOFTWARE TESTING

DISSERTATION

Presented to the Faculty
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 Doctor of Philosophy

Camdon R Cady, BS, MS
Capt, USAF

September 2017

AFIT-ENG-DS-17-S-005

A TREE LOCALITY-SENSITIVE HASH FOR SECURE SOFTWARE TESTING

DISSERTATION

Camdon R Cady, BS, MS
Capt, USAF

Committee Membership:

Gilbert L. Peterson, PhD
Chairman

Douglas D. Hodson, PhD
Member

Mark E. Oxley, PhD
Member

Michael R. Grimaila, PhD
Dean's Representative

ADEDJI B. BADIRU, PhD
Dean, Graduate School of Engineering and Management

AFIT-ENG-DS-17-S-005

Abstract

Bugs in software that make it through testing can cost tens of millions of dollars each year, and in some cases can even result in the loss of human life. In order to eliminate bugs, developers may use symbolic execution to search through possible program states looking for anomalous states. Most of the computational effort to search through these states is spent solving path constraints in order to determine the feasibility of entering each state. State merging can make this search more efficient by combining program states, allowing multiple execution paths to be analyzed at the same time. However, a merge with dissimilar path constraints dramatically increases the time necessary to solve the path constraint. Currently, there are no distance measures for path constraints, and pairwise comparison of program states is not scalable. A hashing method is presented that clusters constraints in such a way that similar constraints are placed in the same cluster without requiring pairwise comparisons between queries. When combined with other state-of-the-art state merging techniques, the hashing method allows the symbolic executor to execute more instructions per second and find more terminal execution states than the other techniques alone, without decreasing the high path coverage achieved by merging many states together.

Table of Contents

Page

Abstract iv

List of Figures vi

List of Tables viii

I. Introduction 1

1.1 Introduction 2
1.2 Problem Description 3
1.3 Methodological Approach 4
1.4 Research Questions 4
1.5 Summary 5

II. Background 6

2.1 Program Testing 7
2.2 Dynamic Symbolic Execution and the Path Explosion Problem 16
2.3 Efficient State Comparison 21
2.4 Summary 25

III. Methodology 27

3.1 Path Constraints 27
3.2 Hashing 29
3.3 Example 34
3.4 Implementation 40
3.5 Summary 40

IV. Results 41

4.1 Hash Accuracy 41
4.2 Use in Symbolic Execution 44
4.3 Summary 54

V. Conclusion 56

5.1 Summary 56
5.2 Future Work 57
Appendix A: State Merging Simulation 59
Appendix B: Selecting Symbolic Input 61
Bibliography 64

List of Figures

Figure Page

1 Dynamic Symbolic Execution. 17

2 Tree Locality-Sensitive Hashing (TLSH) Algorithm Diagram. 29

3 A flow graph generated from the query $\{\phi_1, \ldots, \phi_n\}$. 32

4 Simple C Program. 35

5 LLVM Bitcode of the `get_sign` function. 36

6 Flow graph for the path constraint `x == 0`. 37

7 Flow graph for the path constraint `x != 0` and `x < 0`. 38

8 Flow graph for the path constraint `x != 0` and `x !< 0`. 39

9 A spanning tree found within the flow graph of Figure 8. 39

10 Average Tree Edit Distance Between 1000 SMT Constraints. 42

11 Instructions Executed in 60 Minutes Using Naive Merging vs TLSH 47

12 Instructions Executed using TLSH vs TLSH & QCE Combined. 48

13 Instructions Executed using QCE vs TLSH & QCE Combined. 49

14 Instructions Executed using Baseline (No Merging) vs TLSH & QCE Combined. 50

15 Instructions Executed in `nice`. 51

16 Terminating States using Baseline (No Merging) vs TLSH & QCE Combined. 51

17 Terminating States using Naive Merging (All Merges) vs TLSH & QCE Combined. 52

Figure Page

18 Terminating States using TLSH vs TLSH & QCE Combined. 53

19 Terminating States using QCE vs TLSH & QCE Combined. 53

20 Estimated Paths Explored using QCE vs TLSH & QCE Combined. 54

21 Average Speedup for each Coreutil. 59

22 Running Time to Execute `nice` with 2-byte Arguments. 62

23 Paths Explored in 60 Minutes. 63

List of Tables

Table Page

1 Hash Accuracy for Selected Utilities. 43

2 Merging Strategies Tested. 45

3 Comparison between Z3 and STP for Seq and Pathchk. 60

A TREE LOCALITY-SENSITIVE HASH FOR SECURE SOFTWARE TESTING

I. Introduction

Advances in computational power and search algorithms have allowed for the creation of tools that automatically test software by making a model of a program, then using that model to find inputs that cause the program to enter an anomalous state (such as accessing a null pointer or writing beyond the end of an array.) As testing continues for greater periods of time, both the number of program states that must be tracked and explored as well as the difficulty of analyzing each state increase, so that testing can begin to "bog down."

This dissertation presents the *Tree Locality-Sensitive Hash* (TLSH), a locality-sensitive hashing algorithm that with high probability assigns the same hash value to trees that are close to each other under tree edit distance. When applied to the path constraints of symbolic execution states, TLSH identifies similar states for merging to reduce the search space that must be analyzed, thereby increasing the odds of finding a bug/increasing the certainty there there are no bugs remaining in the program under test. TLSH assigns the same hash value to similar program states, so that multiple states with the same hash value can be merged together and analyzed simultaneously. Combining TLSH with Query Count Estimation (QCE) [36] yields performance improvements in instructions executed per second and the number of terminal execution states discovered.

1.1 Introduction

We test software for many reasons, such as to check that it actually does what we intended it to do, to see if users are able to successfully use the software in order to accomplish their goal, to make sure that updates to the software do not introduce errors, or to look for vulnerabilities that could allow a nefarious actor to use our software against us. Ultimately, all testing is designed to find errors in software so that they can be corrected [45]. Software errors have a real cost [43]: human lives have been lost due to software bugs, and an estimated 60 billion dollars a year are lost in the US economy every year due to software errors.

Software testing is an important and expensive part of the software development lifecycle [45]. Testing occurs throughout the lifecyle of a piece of software, from unit testing that tests individual modules to higher order testing of entire systems of interrelated software. For each testing stage, test inputs must be generated. These inputs may be created by the developers writing the code, individuals responsible for testing the code, or even automatically by a testing framework.

Symbolic execution is a popular method to generate test inputs for complex pieces of software [14]. Symbolic execution is a program analysis technique in which concrete inputs in programs are replaced with models representing the possible values those inputs may take on, so that the execution path of any possible input may be evaluated, as opposed to a single execution path. A subset of program inputs are marked as *symbolic*. As the program is symbolically executed, data that is dependent on a that input is also recorded as symbolic. When a branching instruction depends on a symbolic value, the feasibility of taking each branch is evaluated by querying a Satisfiability Modulo Theories (SMT) Solver [49]. A symbolic program state is created for each of the possible branches.

1.2 Problem Description

As symbolic execution continues, the number of states that are discovered (and thus, the number of paths to be explored) grows exponentially in the number of branches encountered; this is known as the *Path Explosion Problem* [33]. In general, it is not possible to explore every possible execution path (the program may not halt for every possible input). One method of mitigating the path explosion problem is to use heuristics to select states for further exploration that are more promising, such as states that have a higher probability of increasing code or branch coverage [12]. This can result in high code coverage, but still "bogs down" when testing complex real-world applications. Another method is to merge similar states [28, 36]. Merging states can alleviate the path explosion problem by reducing repeated work, but selecting the wrong states to merge can result in a decrease in code coverage due to an increase in the difficulty of solving the path constraints of merged states [18]. These approaches can be used concurrently by using heuristic exploration to select new states to execute and merge as appropriate.

Finding the correct states to merge is not trivial [36, 18]. A naive approach to selecting states to merge is to define a distance measure between the states and compute the distance between each state, and finally merge states that are close together. For the tests described in this document, the symbolic execution tool frequently had hundreds of thousands of states that were awaiting exploration, which would result in billions of comparisons required to make a merge decision. An alternative approach is to hash the states in such a way that states that are similar hash to the same value. This avoids the cost of the pairwise comparisons while separating the states into sets that are appropriate for merging.

1.3 Methodological Approach

The novel hashing technique to compare path constraints involves representing each path constraint as a graph, enumerating certain k-subgraphs of that graph, and calculating the canonical form of each of those subgraphs. Min-hash [11] is then used to calculate a hash of each path constraint based on the subgraphs present within it. Since this hashing technique assigns with high probability the same hash value to constraints that are close to each other under tree edit distance, we call it *Tree Locality-Sensitive Hash* (TLSH).

To test TLSH, the GNU Coreutils [2] were symbolically executed. Two sets of experiments were performed. In the first, the Coreutils were symbolically executed with no state merging, and the queries sent to the solver were logged. These queries were then used to evaluate the accuracy of the hash compared to tree edit distance. In the second set of experiments, the Coreutils were symbolically executed using several different state merging strategies, and the total number of instructions executed, the number of execution states executed to completion, and the estimated number of partial paths in the CFG were compared for each merging strategy.

1.4 Research Questions

Analysis of TSH must answer two questions: is it likely to assign the same hash to a constraint and its nearest neighbor, and does it cluster path constraints in a way that is useful for state merging?

To answer first question, five representative utilities were selected, and the tree edit distance [52] between every constraint generated during the testing of a utility computed. Over 99.3% of the constraints are assigned the same hash value as at least one nearest neighbor.

To answer the second, the constraint hashing algorithm was implemented in Klee

[12], and the GNU Coreutils [2] were symbolically executed for an hour each using various state merging strategies. A merging strategy that combined TLSH with Query Count Estimation (QCE) [36], a technique to minimize the number of calls to the solver, resulted in the most instructions per second executed, the most execution states executed until they halted, and an equivalent number of feasible paths discovered by Klee.

1.5 Summary

This document the *Tree Locality-Sensitive Hash* (TLSH), a locality-senstive hashing method that assigns a hash value to the path constraints generated during symbolic execution. This hashing method assigns the same value to path constraints that are close to each other under tree edit distance. Constraints with the same hash value can be combined together and solved quickly, which allows Klee to examine more execution states than it can using other strategies.

Chapter 2 describes the motivation for TLSH, and covers graph comparison and locality-sensitive hashing techniques relevant to its development. Chapter 3 describes TSH in detail. Chapter 4 gives the results of performing two groups of tests that verify the accuracy and usefulness of TLSH. Chapter 5 summarizes the contents of the dissertation and lists avenues for further related research.

II. Background

All but the most trivial programs (e.g., Hello, World) process some sort of input. This input may come from text entered by the user, a file read from a disk, bytes from a network socket, or a multitude of other sources. Test inputs are created to test the behavior of these programs. For most programs, it is not feasible to test every possible input: consider a program that reads just 8 bytes of data and therefore has 2^{64} possible inputs. Assuming (rather unrealistically) that we can test a billion inputs each second, it would take approximately 585 years to test all of the possible inputs, and that doesn't even include the time necessary to actually run the program for each input and analyze the output for errors. Clearly, only a small subset of the possible program inputs can be used to test the program.

The work presented here focuses on a particular solution to automatically generate a small set of test inputs that do a "good" job of testing a piece of software: symbolic execution [14]. In symbolic execution, the tester specifies that some input is symbolic. As the program is symbolically executed, the possible range of values that that input (and any data that is dependent on that input) can take on is tracked as each instruction or line of code is executed. When program execution terminates, or when an anomalous state is encountered (such as a reference to a null pointer), then a concrete value in the range of possible values is computed, and that value is saved as a test case. Symbolic execution can find test cases that exercise bugs that would likely never be found using random inputs or even inputs generated by developers and users. To reduce the cost of performing many expensive (in terms of processor time and memory required) computations in order to determine what values a symbolic variable can take on, this dissertation explores a technique, state merging, that reduces the number of computations required, without making those computations so much more expensive as to outweigh the benefits fewer computations.

2.1 Program Testing

Most software is subjected to some sort of testing before it is released [45]. *Verification testing* applies a small number of inputs to a program to ensure that the program complies with requirements. *Unit testing* checks that individual modules meet their specifications. *User testing* analyzes whether the program can be successfully used and understood by the target user base.

However, it is not sufficient that a program simply work properly for all "good" inputs, but that it fails gracefully when presented with a "bad" input. Consider the anecdote presented by Miller [44] as motivation for their automated UNIX utility tester: a user is logged in to a workstation via a dial-up connection when bad weather starts generating all sorts of noise on the telephone line, permuting the commands being sent. No sophisticated user would expect utilities to work properly when presented with junk input, perhaps the most that could be expected would be a helpful error message about what went wrong. To the authors' surprise, the programs didn't simply terminate, but many would crash entirely, indicating the possible presence of serious bugs.

2.1.1 Test Concepts.

Software testing can be partitioned into two broad categories [45]: black-box testing and white-box testing. Black-box testing does not utilize any information about *how* the program operates, only program inputs and outputs. White-box testing uses information about the logic or structure of a program to test it.

In order to exhaustively black-box test a program, every possible input must be applied [45]. For programs with non-trivial input space, this is not possible. In order to select a small number of inputs to apply to the program while maximizing the chance of finding bugs, strategies such as partitioning the input space into equivalence classes and testing at least one member of each class [63], testing inputs that are boundary values [63], or cause-effect graphing (a method of formalizing the definition of a piece of software and using that formal definition to generate test inputs) [45] are utilized.

Because the structure of a program is used in a white-box tester, a program may be exhaustively tested by exercising every possible execution path through the program [45]. While this likely requires significantly fewer inputs than exhaustive black-box testing, it is still not feasible for many programs with loops.

In order to quantify how well a set of inputs exercises a piece of software, the concept of coverage is often used [27]. *Statement coverage* or *line coverage* is the percentage of statements or lines of code that are executed when the test inputs are applied. Because of the intricacies of code, statement coverage often fails to detect errors. A stronger measure of coverage is *branch coverage*, which is the percentage of branches in the program that are taken. In order to achieve 100 percent branch coverage, each conditional must evaluate to true for some test case and false for some test case.

2.1.2 Random Black-Box Testing: Fuzzing.

Fuzzing is a black-box testing strategy where inputs are randomly generated. Miller, et al. [44] developed a random input generator for UNIX command-line utilities that they dubbed a "fuzzer." When used as part of a framework that applies the inputs to utilities and then monitors their execution to see if the program crashes and logs the results, the fuzzer discovered a significant number of bugs: between a quarter and a third of the utilities tested crashed on any given system.

Because of the simplicity of generating random inputs, a number of fuzzing frameworks have been developed since Miller's fuzzer was introduced, and fuzzing has become a fairly common method of testing for bugs [59]. Eventually though, the "low-hanging fruit" will have been picked, and simply fuzzing for longer isn't likely to lead to better code coverage or discovery of more bugs [25].

2.1.3 White-Box Fuzzing.

If the goal of an automated test system is to enhance coverage, then a priority should be placed on generating test cases that cover new lines or branches of code. Similarly, if the goal of testing is to discover an exploitable vulnerability, then there are certain pieces of code that are likely to contain exploitable bugs, such as calls to `strcopy` [7], and the goal should be to generate test cases that exercise that code. In either case, specific sections of code can be targeted for execution.

Taint analysis [7] tracks dependencies between pieces of data at different locations in a program. By propagating the dependencies of each piece of input, the fuzzer can determine what sections of the input must be modified to change execution of the program under test. This approach provides more coverage than other fuzzing techniques, while still being relatively fast.

Taint analysis does not guarantee that the fuzzer can find an input to reach the targeted code (because the fuzzer may need to try every permutation of the input) [59]. In order to either guarantee that a test case is generated that tests the targeted code section or prove that no test case will test it, a more complete model of the program must be used.

The reason that more fuzzing eventually doesn't lead to better code coverage is that even "small" inputs represent an enormous search space, and some sections of code are only reachable from a tiny sliver of the search space. Consider a conditional (*if-then-else*) statement that checks for equality between a 32-bit integer variable and a constant. Such statements are prolific in code, and certainly doesn't represent a complex or unusual condition. The integer can take on 2^{32} different values. American Fuzzy Lop [1], a highly sophisticated fuzzer, reportedly can execute thousands of inputs per second, so assume that we can apply 10,000 inputs per second. At that speed, searching the entire input space would still take 119 days. Considering that many bugs require an input of more than four bytes to be triggered, it is understandable why fuzzing only catches "shallow" bugs. What is needed is a way to reason about the semantics of the program in order to find an input that satisfies a particular branch condition. This reasoning is accomplished by solving satisfiability problems.

2.1.4 Satisfiability Problems.

The boolean satisfiability problem, frequently referred to as simply the SAT problem, is as follows: given an expression comprised of variables and logical operators, does there exist a consistent assignment of all the variables that results in the expression evaluating to true [21]? SAT is the first known NP-complete problem, and as such, there is no known polynomial-time solution to solve the generalized SAT problem [17]. However, advances in computing power and algorithms have allowed for large instances of SAT to be solved.

The output of a satisfiability solver is either an assignment of variables that simultaneously satisfies all of the formulas (a "model"), or the result UNSAT if the formulas cannot be simultaneously satisfied.

2.1.5 DPLL Solvers.

Most modern SAT solvers are based the work of Davis, Putnam, Logemann, and Loveland [21, 20] who describe a backtracking procedure to compute the satisfiability of an expression. Since each boolean variable must be true or false in any assignment, the DPLL algorithm selects a variable and assumes an assignment. That assignment is propagated through the expression, and the result is an expression with one fewer variables. This expression may contain an inconsistency, in which case the algorithm backtracks and assumes the opposite assignment. If the resulting expression is not inconsistent, and more variables remain, the algorithm is repeated. At the end, the algorithm either terminates with a satisfying assignment (the assumptions made during the successful algorithm traversal), or if all backtracks are taken and it is still not possible to solve the remaining expression, the original expression is not satisfiable. Solvers employing such a backtracking algorithm are known as DPLL solvers after the original authors.

2.1.6 Satisfiability Modulo Theories.

SAT can be extended to include expressions in first-order logic. This allows for expressions to encode more complex problems by including "theories" such as integers, real numbers, or arrays [49]; these are referred to as Satisfiability Modulo Theories (SMT) solvers. Since their introduction, decision procedures integrating multiple theories have grown increasingly advanced; a few of the notable advancement are highlighted here.

Nelson and Oppen [47] proposed a method that efficiently simplifies theorems containing statements from multiple theories by finding equalities in one theory and propagating them to other theories. Examples of equalities that could be propagated are the solution to a system of linear equations or the result of a read operation following a write operation to an array. Model-based theory combination [22], used in Z3 [9], maintains a model for each theory, and attempts to minimize the number of equalities that must be propagated between solvers for different theories.

Simple Theorem Prover (STP) [23], simplifies the problem as much as possible using a linear system solver, then uses a novel refinement loop that heuristically selects a subset of the simplified problem and converts it to a boolean SAT problem. If the SAT problem is unsatisfiable then the entire problem is unsatisfiable, otherwise, STP tests the solution returned by the SAT solver against the remainder of the simplified problem. Any terms that are not satisfied by the solution are added to the subset being sent to the SAT solver. Eventually, a (not necessarily proper) subset is found that is unsatisfiable, or a solution is found that satisfies the entire problem. This allows STP to solve SMT problems that feature large arrays quite quickly compared to Z3. SMT solvers are leveraged in white-box testing in order to reason about programs.

2.1.7 Reasoning About Programs: Symbolic Execution.

In order to discover inputs to reach code that a fuzzer misses, some part of the input to the program can be made symbolic, which allows it to take on any value instead of taking on a single concrete value. As the program executes, a system of constraints on this input that will cause the program to follow a particular path can be tracked. When the program terminates or a bug is encountered, this system of constraints can be solved in order to generate an input that exercises that path. The constraints are solved by a *Satisfiability Modulo Theories* (SMT) [47] solver.

2.1.7.1 Finding Smart Permutations.

One relatively simple symbolic execution technique is to begin with a concrete input, execute the program, and symbolically generate similar inputs that exercise different paths in the program [26]. This technique can be applied to any program, even if the source code is not available. Microsoft developed an internal tool, Scalable Automated Guided Execution (SAGE) [25], that starts with a valid program input and then uses symbolic execution to generate permutations of that input. It does so by first executing the input and logging each instruction that is executed (the *trace*). The trace is then analyzed with part of the input being made symbolic. Whenever a branch is encountered that is dependent on the symbolic input, SAGE attempts to solve for an input that takes the opposite branch that the original input took. This search scheme is dubbed Generational Search. The authors state that starting from a single input, thousands of test cases can be derived in such a manner. It is built on top of the TruScan [46] replay framework for iDNA [8] traces, which dramatically simplifies the task of keeping track of the program state because it provides an interface to useful program information such as symbols information, system calls, memory allocations, and data flow. To solve the SMT constraints generated during symbolic execution, SAGE relies on the Z3 solver [9]. Since it's development, SAGE has discovered numerous security vulnerabilities and other bugs in Microsoft products [25].

Another tool that works in a similar manner is Pathgrind [3], which builds upon the widely-used Valgrind [48] program-checking suite to symbolically execute binaries. Valgrind instruments native code and executes it, automatically checking for a number of different errors. Valgrind generates an intermediate form of the program being checked, and Pathgrind symbolically executes this intermediate form. Like SAGE, Pathgrind also starts with a valid input, and then symbolically executes the program to find permutations of that input that cause the program to take different execution paths. Pathgrind uses Simple Theorem Prover (STP) [23], a theorem prover designed for the large conjunctions typically encountered in path constraints.

2.1.7.2 Symbolic Source Code Analysis.

SAGE [25] and Pathgrind [3] can enhance code coverage and find bugs with minimal additional work by the developer; all they require is an executable program and a single valid input to produce a multitude of new test cases. While these tools are able to quickly generate a number of interesting permutations of existing concrete inputs, they are just modifications of inputs that are supplied by the tester, and not a search through the multitude of other execution paths through the program. In order to more exhaustively search the state space of a program, a more substantive model of the program's behavior can be developed. A couple of approaches that proved to be useful create a model of the program directly from the source code. Access to source code can simplify some aspects of symbolic execution because of the broad array of information directly available, such as symbol type and program structure.

EXE [13] and Directed Automated Random Testing (DART) [57] both analyze high-level source code (C code) and generate inputs to drive program testing. Some input is marked as symbolic, then for each path in the program the constraints required to follow that path are collected, and an SMT solver is used to check if the path is feasible, and if so, generate a test case. Both tools are capable of automatically detecting errors such as failed assert statements, program crashes, memory out-of-bounds, etc.

EXE [13] uses Simple Theorem Prover (STP) [23] to solve the constraints generated during symbolic execution. STP showed a significant improvement compared to the state-of-the-art solver at the time, the Cooperating Validity Checker Lite (CVCL) [6]. EXE discovered previously unknown bugs in the packet filters used in FreeBSD and Linux (in code that had been in use years without the bugs being discovered), in the udhcpd DHCP server, and in a commonly used regular expression library.

DART [57] was able to generate inputs for a majority of external functions in the oSIP, an open-source Internet-Protocol (IP) telephony library, that crashed the program, including finding at least one bug that was considered a significant security vulnerability. The theorem prover (SMT solver) used by DART is not revealed in the paper.

2.1.7.3 Modified Virtual Machines.

High-level languages can be compiled to an intermediate representation, often called bitcode or bytecode, instead of natively executable code. This can allow for operations that are likely to result in bugs, such as bounds checking, to be moved into a runtime that can be validated before release (as Microsoft's .NET framework does), or allow for a single optimizer and code generation engine to be used with a multitude of compilers (as in the LLVM project [37]). These intermediate representations are more compact than the source code, are portable(they can be executed on any platform with the correct runtime engine), and often contain more type information than is embedded in an executable binary for x86 architecture. Because a virtual machine already exists that can process the intermediate form, these virtual machines can be modified to symbolically execute the same code.

Pex [61] is a test-generation tool for .NET programs that utilizes symbolic execution to generate a small test suite with high coverage for a .NET program. When used to generate tests for an extensively-used .NET component, Pex was able to find a number of previously undiscovered bugs, including a number of assertion violations. Pex can analyze any .NET component, regardless of the language it was written in, but the associated Visual Studio plugin only supports C#.

Klee [12] is a symbolic execution tool that executes LLVM bitcode, along with a set of libraries (klee-uclibc) to allow common inputs (files, stdin, etc) to be made symbolic. When it was introduced, Klee was used to enhance test coverage and find previously undiscovered bugs in the coreutiles [2] library, and to uncover functional differences between Busybox [62] and Coreutils utilities that were supposed to function identically. Klee is open source, remains under active development, and is used to test the ideas put forth in this document.

2.2 Dynamic Symbolic Execution and the Path Explosion Problem

In dynamic symbolic execution, a path through the program is executed until it terminates or an error condition is found [59]. In practice, this is implemented by keeping track of a pool of execution states, each one containing, at a minimum, the address of the next instruction to be executed, expressions representing registers and memory that are in scope, and a path constraint that describes the conditions that cause the program to traverse the same execution path that the execution state has executed thus far [28]. A state is selected and executed for some time slice, or until it halts. When branches in the program are encountered, new states are created and added to the pool. When a state halts, it is removed from the pool. Symbolic execution continues until all states have terminated, or until a time limit is reached. This process is illustrated in Figure 1.

When a branching instruction is encountered during symbolic execution, each feasible path of the branch is explored. As the program symbolically executes and branches are encountered, there is the possibility for exponential growth in the number of program paths to be analyzed [14]. If a program does not halt, then all possible paths through the program cannot be enumerated or explored. This is referred to as the *Path Explosion Problem.* The path explosion problem can be mitigated by *prioritizing* the symbolic execution of promising paths, analyzing a program modularly to decrease the number of branching instructions encountered, or *merging* execution states together [59].

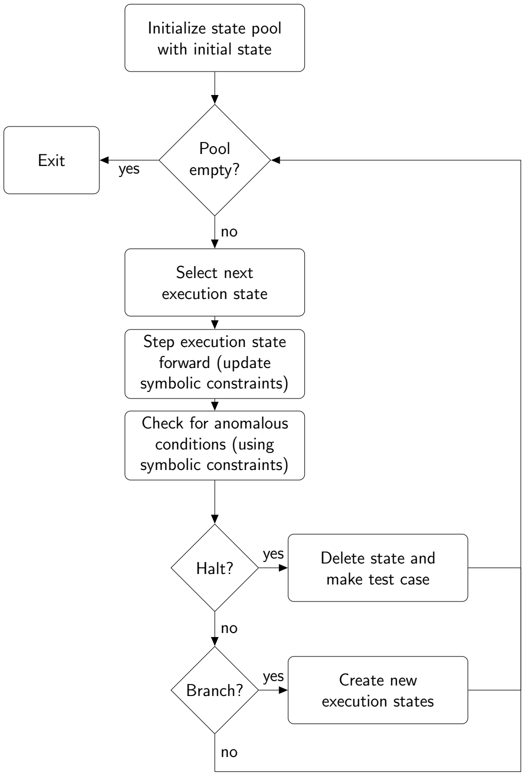

Initialize state pool with initial state
Pool empty?
yes
Exit
no
Select next execution state
Step execution state forward (update symbolic constraints)
Check for anomalous conditions (using symbolic constraints)
Halt?
yes
Delete state and make test case
no
Branch?
yes
Create new execution states
no

Figure 1. Dynamic Symbolic Execution.

2.2.1 Prioritizing Promising Paths.

High code coverage can be achieved by selecting the most promising paths and symbolically executing those. Klee [12] interleaves random path selection, which "opens up" areas of the program that have not been symbolically executed yet, and a heuristic search to select states that are more likely to cover new code. This allows Klee to achieve high code coverage in less time than random search alone. Klee achieves higher code coverage than the test suite for Coreutils [2], which is a mature and highly-tested piece of software.

Li, et al. [39] use subpaths of length n as a heuristic to perform subpath-guided path exploration, which is a search technique that prioritizes "less traveled" areas of the program being tested in order to find bugs. Klee [12] found bugs when using subpath-guided path exploration that were not found using other search technique.

Prioritizing more promising execution states to explore still leaves the searcher with an exponentially growing number of states to choose from. Given the finite resources available to explore paths, the symbolic executor will still be forced to discard or ignore the vast majority of paths (see Appendix B for how the path explosion overwhelms the symbolic executor), and heuristics don't guarantee that no bugs lie on those unexplored paths.

2.2.2 Modularly Testing Programs.

The difficulty of symbolic execution generally increases as the size of the program being tested grows. By testing a program modularly the path explosion problem can be mitigated. SMART [24] extends DART [57] by testing functions individually and generating function summaries. When a function is called with symbolic values, the summary is used, instead of exploring potentially hundreds of paths through the function.

Concolic testing [58] combines concrete and symbolic execution to mitigate the difficulty of having purely symbolic variables. Traditional concolic testing performs concrete and symbolic execution in parallel, and replaces any unsolvable constraints encountered during symbolic execution with the concrete values taken on during concrete execution. Hybrid concolic testing [40] interleaves concolic and random testing. Testing begins with random testing (fuzzing). When the fuzzer doesn't uncover new statements for a predetermined number of steps, the algorithm switches to concolic execution until an uncovered statement is encountered, at which point more random testing is performed.

Lazy expansion [41] builds on concolic testing by assuming that low-level library functions have already been tested. When a call to a library function is encountered during concolic execution, instead of entering and symbolically executing the library functions the called function is replaced by an unconstrained input, and concolic execution continues. The concolic execution algorithm will generate a number of execution paths, including some that may not be feasible due to the replacement of the function call with an unconstrained input. To determine the feasibility of executing these paths, the algorithm only has to find a path in the library function that satisfies the constraints encountered during concolic execution. By only searching for a relevant path in the library function, the algorithm avoids spending unnecessary time testing the library function.

Klee [12] is able to execute instructions that don't reference symbolic data at "native" speed (native to the LLVM virtual machine, not the underlying hardware), and so interleaving multiple search strategies in Klee is able to achieve some of the benefit of hybrid concolic testing. Lazy expansion requires that certain libraries be abstracted, this works well for programs that rely only on a small number of libraries, or perhaps only on system calls, but in a complex program with many library dependencies, there will still be an exponential explosion in the number of paths to be explored. In order to actually reduce the explosion, states must be eliminated from consideration somehow. State merging accomplishes this reduction in the number of execution states under consideration.

2.2.3 State Merging.

Another method for alleviating the path explosion problem is to explore multiple paths simultaneously. Babi and Hu [5] reduce the time to solve multiple SMT queries by representing verification conditions (SMT queries) as directed acyclic graphs (DAGs), then reusing "facts" learned solving one VC when solving another VC. The DAGs used to represent the VCs are similar to those used in this paper.

If a program arrives in an identical execution state via two different execution paths, then both of those states will produce identical results going forward, and only one of them needs to be kept [10]. RWset (short for Read/Write set) expands on this concept. If two execution states differ only in variables that are never read again, they will also produce identical results going forward and only one of them needs to be executed. Discarding redundant paths can reduce the number of paths explored by up to an order of magnitude, while achieving the same coverage.

State merging reduces the number of paths to explore by merging similar execution states, essentially merging two or more execution paths together and executing those paths simultaneously for the remainder of symbolic execution [28]. Kuznetsov, et al. [36] compute a query cost estimate (QCE) for a pair of symbolic execution states that is a heuristic estimate of the computational effort that will be required in order to symbolically execute the two states if they are merged or kept separate [36]. This estimate is then used to define a similarity relation between two states, and the states are merged if they are similar. Their similarity heuristic uses information gleaned by pre-processing the CFG of a program to be symbolically executed. The QCE attempts to limit the increase in calls to the SMT solver caused by merging states. Opportunistically merging states whenever the possibility presents itself does not necessarily lead to enhanced code coverage [18]. This is due in part to the fact that some state merges result in a path constraint that is more difficult to solve, and so any speedup gained by exploring fewer paths is lost to extra time spent solving the path constraint. This dissertation presents a method by which the path constraints can be efficiently compared, so that merges that significantly increase the difficulty of symbolic execution can be avoided.

2.3 Efficient State Comparison

There are three challenges encountered when attempting to compare the path constraints of execution states during symbolic execution:

1. Due to the path explosion problem, there are often 100,000 states or more in the state pool. Comparing each in a pairwise manner results in an unfeasible number of comparisons being required.

2. A good similarity measure for SMT constraints has not previously been published.

3. SMT and SAT problems lend themselves to representation as trees, where interior nodes of the tree are some sort of operator and leaf nodes are constants or variables. The best published methods for comparing trees are still quite slow for trees of the size encountered during symbolic execution; computing the tree edit distance between 1000 trees generated from SMT queries during symbolic execution takes between 12 and 24 hours.

In order to develop a comparison technique that overcomes these challenges, this section examines hashing techniques that facilitate finding similar states without performing pairwise comparisons, similarity measures used to find programs that are similar to each other, and graph comparison techniques.

2.3.1 High-Collision Hashing.

Traditional hashing techniques for file integrity checking and cryptographic verification are designed so that changing a single bit in the file being hashed changes the digest [53]. At the other extreme, hash functions for clustering and retrieval have been designed such that inputs that are similar (for some definition of similarity) have a high probability of hashing to the same digest.

Existing locality-sensitive hashing techniques include Min-hash [11], which takes a sample of two sets S_1 and S_2 and estimates an indicator of similarity called the *Jaccard Index*, given by ${}^{|S_1 \cap S_2|}/_{|S_1 \cup S_2|}$. Locality Sensitive Hashing (LSH) [30], which reduces the dimensionality of high-dimension data, is leveraged for approximate nearest neighbor problems [4] in Euclidean space, and is applicable to other domains as well, including the set-similarity problem addressed in [11]. Researchers in digital forensics and spam detection have developed *Context Triggered Piecewise Hashes*(CTPH), hashes that only consider part of an input [34].

2.3.2 Program Recognition and Similarity.

The problem of recognizing similar path constraints is similar to recognizing pieces of code that are similar to each other. Recognizing when two pieces of code are similar is useful for detecting common erroers, or for attributing a piece of software to an author. Rich and Willis [55] generate a flow graph representing a piece of software, then search for known subgraphs (corresponding to cliches in the library) within that graph. In order to recognize the subgraphs, a context-free grammar for generating graphs is used.

Collberg and Nagra [16] outline several methods of analyzing the similarity of two software programs for the purpose of software birthmarking, software forensics, plagiarism detection, and clone detection. In order to detect software plagiarism or cloning, the program dependence graph (PDG) of a function is generated. A PDG is a flow graph in which the vertices represent statements within a software program and and edges represent a dependence of one statement on another. The detection methods then search for identical or similar subgraphs within 2 PDGs in order to detect plagiarism or software cloning. A birthmarking method is presented that first computes metrics on a program based on the layout, style, or structure of a program, and attributes that program to a particular author based on a vector formed by those metrics.

Other program recognition methods focus on the detection of malware. These techniques attempt to be robust against obfuscation methods, and fast enough to be run in “real-time,” as in anti-virus tools used to protect personal computers. Cesare and Yang [15] present two different methods of generating a feature vector of a program from its control flow graph (CFG), one based on k-subgraphs within the CFG and one based on Q-grams based on a canonical string representation of the CFG. They then compare the feature vectors to feature vectors of known families of malware. Their work builds on that of Kruegelm, et al. [35], who fingerprint network communications containing executable code in search of self-propagating malware. Jin, et al. [31] classify functions within a program in order to associate them with a library of known functions. They first create a hash associated with each basic block within the function, then apply the Min-hash algorithm [11] to create a hash of each function. Due to the properties of Min-hash, similar functions hash to the same value. While these hashes take into account the input-output behavior of each basic block, selecting states for state merging benefits from taking into account which program data the functions were applied to as well [36].

One commonality between many of these methods of recognizing similar programs is that they analyze some sort of graph that captures relevant information about the program. In the following subsection, the idea of a graph is formalized, and what it means for two graphs to be similar is discussed.

2.3.3 Graph Comparison.

A *graph* is a set of vertices and edges, where each edge has two endpoints that are vertices. If the endpoints of the edges are ordered, then the graph is *directed*, otherwise is it *undirected*. If each edge has two distinct endpoints, then the graph is *simple*. A connected, undirected graph with precisely one fewer edges than vertices is a *tree*, which may be *rooted* or not.

2.3.3.1 Graph Isomorphism.

Two graphs are identical if there is a bijection between the nodes of the two graphs such that the connectivity in the graph is maintained under the bijection. Determining whether or not two graphs are identical is known as the *graph isomorphism problem* [19]. While it has not been shown to be a member of a complexity class such as NP-complete, there currently does not exist a polynomial-time algorithm to determine graph isomorphism in the general case. As an upper bound, all 2^n possible bijections must be tested (where n is the number of nodes in each graph.)

Nauty [29] and Bliss [32] are tools designed to compute canonical forms of a graph. Two identical graphs will have the same adjacency matrix when placed in canonical form. Since these do solve the graph isomorphism problem, none of these algorithms work in polynomial time, but they are software that is generally quick and stable for fairly small graphs.

2.3.3.2 Graph Similarity.

Zager and Verghese [64] developed an algorithm for evaluating the similarity of two graphs. Their algorithm computes the pairwise similarity between each vertex and edge in two graphs. It is based on the intuition that two vertices or edges are similar if their neighborhoods are similar. The similarities are iteratively computed until they converge.

Another measure of the similarity between graphs, in particular trees, is the *edit distance* between them. Operations such as node insertion, removal, or renaming are assigned a cost, the minimum edit distance is the cost of the cheapest sequence of operations that transforms one tree into another. RTED [50], the Robust Tree Edit Distance, and it's successor APTED [51, 52] are algorithms for computing the minimum edit distance between trees. Extensively-used implementations of both are provided by the authors.

Other techniques to compare graphs utilize the concept of a *graph kernel*, in which structures of a graph are extracted and then fingerprinted, and graphs are compared using their fingerprints [60]. Graph kernels have been used on databases of chemical structures [60, 54], in combination with machine learning to classify a compound based on known compounds, or to retrieve similar graphs from the database. Of particular interest, Teixeira, et al. [60] use random walks in a graph as kernels, and then use Min-hash to cluster the chemicals without having to directly compare the graphs.

Manzoor, et al. [42] describe a technique that utilizes locality-sensitive hashing on flow graphs, which defines a k-shingle on a graph, then uses hashing techniques to cluster the flow graphs. The dataset used was a set of flow graphs generated by capturing system calls while either benign or malicious activity was taking place on a computer, with the aim of detecting malicious activity.

2.4 Summary

The purpose of testing is to find bugs (errors) in software. It is infeasible to test a program by applying every possible input and observing that the program exhibits appropriate behavior, so a limited number of inputs must be selected to test the program. One way to automatically generate those test cases is symbolic execution, where certain inputs are marked as symbolic and can take on any value during execution. Symbolic execution is capable of "reasoning" about *any* feasible path through the control-flow graph (CFG) of a program, where a single concrete input only exercises a single path through the program. The number of paths through the CFG grows exponentially as more branching instructions are executed; this is known as the *path explosion problem*. One way to mitigate the path explosion problem is to merge similar execution states together, which allows the symbolic executor to explore multiple paths at once. Not all merges are equally good: merging some states together results in path constraints that take much longer to solve, negating any benefits gained from the merge.

This chapter presented a rationale for generating a small set of inputs to test a program or function, several methods of varying sophistication and complexity for generating those inputs, challenges encountered while performing one of those methods (symbolic execution), and related work in the fields of high-collision hashing, program recognition, and graph comparison. The remainder of the document builds upon this foundation and presents a method of comparing the path constraints in order to decide whether states should be merged together. The path constraints are naturally represented as graphs, and the direct comparison of graphs to evaluate their similarity takes a prohibitive amount of time. Various approaches to estimate the similarity of two graphs are explored, and the remainder of this dissertation presents and analyzes a novel technique to compare path constraints.

III. Methodology

To select states to be merged without having to compare each of them, we present a Tree Locality-Sensitive Hashing (TLSH) technique that quickly identifies path constraints that have a similar structure. By selecting states with similar path constraints to be merged, we avoid the computational penalty seen in previous state merging efforts [18] that is caused by increasing the difficulty of a path constraint when merging states, without the use of computationally expensive comparators such as tree edit distance [52].

Hashing a path constraint is done by converting a SMT constraint into a flow graph, finding a spanning tree in that flow graph, using the spanning tree to find k-subgraphs in the graph, finding the canonical representation of each k-subgraph, and finding a min-hash digest based on the unique subgraphs. The form of path constraint used by Klee [12] is described, then each step of the hashing process is presented, and finally an example is given that describes how a simple C program is symbolically executed and the path constraints are hashed.

3.1 Path Constraints

Satisfiability Modulo Theories (SMT) solvers have been extended to solve a myriad of satisfiability problems from different domains [47]. We focus on only constraints that are composed of bit vectors and arrays, which are a natural way to encode the path constraints generated during symbolic execution. Klee[12] further narrows the range of constraints used by only creating constraints in a canonical form. This simplifies the task of recognizing common elements between constraints.

Definition. An SMT constraint is an ordered set of quantifier-free bit vector logic expressions. Each expression is represented as a rooted tree. Constraints are generated and stored in a canonical form (described below). A constraint is considered satisfiable if and only if every expression in the set is simultaneously satisfiable; in other words, the constraint is the conjunction of the expressions comprising it.

Internal nodes of a tree representing an expression are operators, and child nodes are arguments to their parents. Note that the result of one operation may be an argument to another operation. For example, two bit vectors could be added together, and the result of that addition compared to a third bit vector.

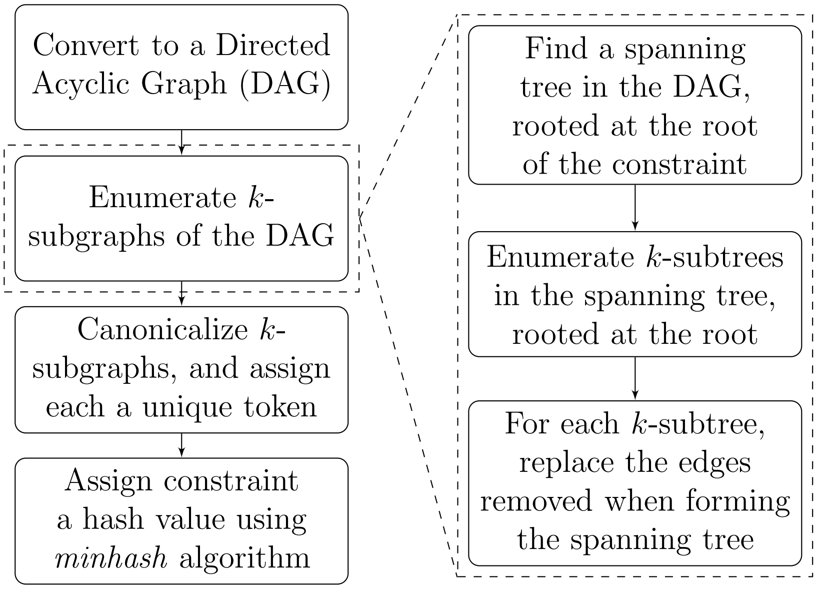
Canonicalization. An expression is said to be **canonical** if it is well-formed and obeys the following rules:

1. No expression has all constant arguments.
2. For boolean expressions:
   - Boolean not is written as $(false ==?)$.
   - $\neq$, $\geq$, and $>$ are not used.
   - Only $\wedge$, $\vee$, $\oplus$, and $==$ logical operations are used.
   - The only boolean operation which may involve a constant is $(== false)$.
3. For any subtree representing a formula, a constant term must be on the left hand side of the root node of the subtree. For example, subtraction by a constant c is written as $add(-c, \ldots)$, not $add(\ldots, -c)$.
4. Chains are unbalanced to the right.

Ordering. The expressions in each constraint are generated from an instruction trace of an executable being symbolically executed. The ordering of the expressions in a query is preserved during the generation of the constraint, that is, expressions generated from two different instructions in the trace will appear in the same order in the query as their instructions appear in the trace. TLSH utilizes this ordering to prioritize the more recent instructions when computing the similarity of the path constraints, so that execution states with long path traces do not end up with an unchanging hash.

3.2 Hashing

Given a path constraint in canonical form, we generate a hash value in such a way that similar constraints are likely to hash to the same value.



Convert to a Directed Acyclic Graph (DAG)
Enumerate k-subgraphs of the DAG
Canonicalize k-subgraphs, and assign each a unique token
Assign constraint a hash value using *minhash* algorithm
Find a spanning tree in the DAG, rooted at the root of the constraint
Enumerate k-subtrees in the spanning tree, rooted at the root
For each k-subtree, replace the edges removed when forming the spanning tree

Figure 2. Tree Locality-Sensitive Hashing (TLSH) Algorithm Diagram.

3.2.1 Conversion to a DAG.

Whereas previous work has typically used the Control Flow Graph (CFG) of a program as the flow graph of interest, this work represents SMT constraints as flow graphs. Flow graphs contain three types of nodes:

Source Nodes which have only edges leaving the node;

Interior Nodes which have edges both entering and leaving the node; and

Sink Nodes which have only edges entering the node.

Intuitively, when a constraint is represented as a flow graph, there is a sink node inserted for each unique leaf node in the constraint tree, a single source node that corresponds to the root node of the constraint tree, and an internal node in the flow graph for each unique internal node in the tree other than the root.

Let an SMT constraint be represented as $\phi = (T, r)$, where graph $T = (E_T, V_T)$ with edge set E_T and vertex set V_T is a tree rooted at vertex $r \in V_T$. For some $x, y \in V_T$, we say that $(x, y) \in E_T$ if and only if there is an edge from x to y in T. Let $children(t)$ be the set of children of a node t, that is, $\{e|(t, e) \in V_T\}$ and the empty set if t is a leaf node.

The flow graph will be represented by the graph $G = (E_G, V_G)$, where E_G is a set of directed edges and V_G is a set of vertices. The functions $add\_vertex(v)$ and $add\_edge((v_1, v_2))$ add a vertex or an edge to G, respectively. Throughout the process of forming the flow graph the algorithm maintains a hash table $H : V_T \to V_G$ that maps nodes in the tree to nodes in the flow graph.

A constraint tree can be added to a flow graph using Algorithm 1.

Note that Algorithm 1, if used on constraints $VC1$ and $VC2$, will generate the *Maximally Shared Graph* described in [5].

Colors can be assigned based on the type of a node(operator, constant, etc). There are finitely many operations defined in any SMT theory, each operation can be given its own color, or operations can be divided into categories and each category can be assigned a color. Klee [12] defines less than 40 types of nodes in SMT trees. In comparison, Kruegel, et al. [35] use 2^{14} colors when coloring flow graphs.

To generate a single flow graph from a set of constraints $\{\phi_1, \ldots, \phi_n\}$, we insert $n - 1$ binary *and* nodes in the graph. How this is done is described in Algorithm 2.

If v_i is the vertex corresponding to the root of constraint tree ϕ_i, then the resulting

Algorithm 1 The function Open

```
function OPEN($t$)
    if $c \in H$ then
        return $H[c]$
    end if
    if $|children(t)| > 0$ then
        $n \leftarrow$ new node
        $H[t] \leftarrow n$
        $add\_node(n)$
        for all $c \in children(t)$ do
            $add\_edge(n,$OPEN$(c))$
        end for
        return $n$
    else
        $v \leftarrow$ new node
        $H[t] \leftarrow v$
        $add\_edge(v)$
        return $v$
    end if
end function
```

Algorithm 2 Generating a flow graph from a query

```
$n \leftarrow$OPEN$(\phi_1)$
for $i = 2, \ldots, n$ do
    $m \leftarrow$ new node
    $insert\_edge(m, n)$
    $insert\_edge(m,$OPEN$(\phi_i))$
    $n \leftarrow m$
end for
```

flow graph looks like Figure 3.

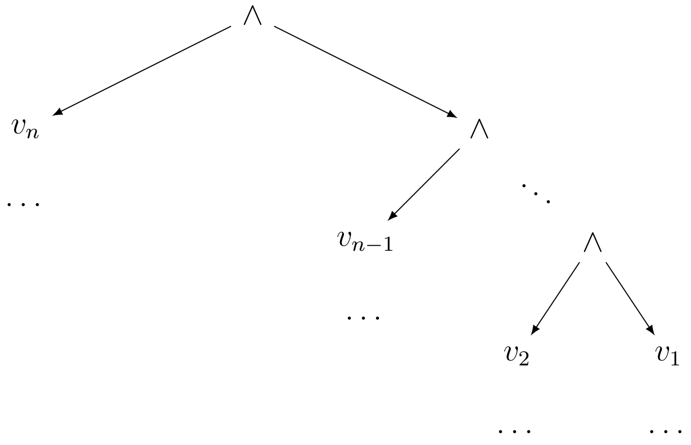


Figure 3. A flow graph generated from the query $\{\phi_1, \ldots, \phi_n\}$.

This places the nodes generated from the last constraint added closest to the source node of the flow graph.

3.2.1.1 More Compact Variants.

Conceptually, the source nodes in the flow graph are some form of terminal node in a constraint, such as a constant value or a symbolic variable. In Algorithm 1, a node is terminal if it is a leaf node in the query. If instead of checking for the number of children in line 2, we check to see if the node is of some terminal type, then the flow graph that is formed will be more compact. As an example, consider a query that has an input array that is indexed using constant indices. Algorithm 1 will create a source node for the array, a source node for each distinct index used, and an interior node for each array read operation. A precise variant would create a single source node for each unique Array/Index pair. An even more compact but imprecise variant would create a single source node for the array and discard the index, capturing the fact that an operation depended on the array but discarding which element of the array it depends upon.

Another useful variant is to replace nodes of the form ($false == \phi$) with a special node of type $\neg\phi$. This not only reduces the size of the graph by replacing 3 nodes with 2, it removes the tree-like structure of the negation, which means that when finding connected k-subgraphs including the root, a subgraph that includes ϕ without the $\neg$ operation is not included.

3.2.2 Finding k-subgraphs.

The features of the flow-graph that we use to form the hash value are k-subgraphs, or subgraphs with precisely k vertices. We use only k-graphs that include the root (source) node of the flow graph, and further require that in order for a vertex to be included in a subgraph, it must be reachable from the source node in the subgraph (there must be a "chain" of vertices from the source node to every other node).

Much like in previous work focusing on CFGs[35, 15], k-subgraphs are generated using a three-step process: a depth-first spanning tree (of height at most k) rooted at the sink node is generated, k-subtrees of that spanning tree that include the root are found, and finally the graphs induced by the vertices in each k-subtrees (each a k-subgraph of the flowgraph) are canonicalized. This method has been fast enough to be useful even in large graphs, but only generates a subset of the k-subgraphs of a graph.

3.2.3 Canonicalizing Subgraphs.

Graphs are canonicalized using the Bliss[32] toolkit. The result of canonicalization is an adjacency matrix. Two isomorphic graphs are guaranteed to have the same adjacency matrix after canonicalization. Each adjacency matrix is converted to a string of length k^2, and these strings are used to generate the hash value of the graph.

3.2.4 Computing the Hash Value.

Min-hash [11] performs a random permutation of a finite set, and then selects the first n elements of the permutation. Those n elements are the hash value for the set. In place of a random permutation, a hash function may be used, and elements of the set sorted by their hash value [38]. The n least elements when sorted this way form the hash value of the set.

The MD5[56] hash of each of the strings found during the previous step is computed, and then the strings are sorted by their hash value. The n strings with the smallest hash value are the signature of the flow graph.

One important feature of the flow graph is that the constraints added to the query more recently are closer to the source node, and therefore are more likely to be included in a randomly selected k-subgraph that includes the source node. Once the query reaches a certain size, it is unlikely that the nodes generated by constraints that appear early in the query will be reached by the spanning tree (because the height is constrained to k). In this sense, the hash functions a bit like a piecewise hash.

3.3 Example

Klee[12] includes several simple examples, one of which is `get_sign.c`. This program declares an integer, and then checks to see whether the integer is zero, positive, or negative. When the integer is made symbolic and the program is symbolically executed, Klee finds 3 paths through the program, one for each of the cases. The example program is shown below. The three execution paths caused by the conditionals on lines 8 and 11. On line 19, the variable `a` is marked as symbolic. Without any symbolic variables, Klee is merely a bitcode executer. Klee also accepts symbolic command-line arguments, symbolic standard input, and symbolic files.

```
 1  /*
 2   * First KLEE tutorial: testing a small function
 3   */
 4
 5  #include <klee/klee.h>
 6
 7  int get_sign(int x) {
 8    if (x == 0)
 9       return 0;
10
11    if (x < 0)
12       return -1;
13    else
14       return 1;
15  }
16
17  int main() {
18    int a;
19    klee_make_symbolic(&a, sizeof(a), ”a”);
20    return get_sign(a);
21  }
```

Figure 4. Simple C Program.

3.3.1 Compilation to LLVM Bitcode.

In order to symbolically execute the program with Klee, it must first be compiled to LLVM bitcode. The disassembly of the function `get_sign` is seen below. The conditional statements on lines 8 and 11 of the C source code are implemented on lines 7-8 and 16-17, respectively.

3.3.2 Symbolic Execution.

The first `if` statement encountered in the program checks to see if the integer `x` is equal to zero, so the path constraint to take the `if` statement is `x==0`. The internal representation of this constraint in Klee is fairly complex, so the flow graph representation has 13 vertices, as shown in Figure 6.

Representing a 32-bit integer value requires 11 nodes in the graph because it is

```
; Function Attrs: nounwind uwtable
define i32 @get_sign(i32 %x) #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 %x, i32* %2, align 4
  %3 = load i32* %2, align 4
  %4 = icmp eq i32 %3, 0
  br i1 %4, label %5, label %6

; <label>:5
; preds = %0
  store i32 0, i32* %1
  br label %11

; <label>:6
; preds = %0
  %7 = load i32* %2, align 4
  %8 = icmp slt i32 %7, 0
  br i1 %8, label %9, label %10

; <label>:9
; preds = %6
  store i32 -1, i32* %1
  br label %11

; <label>:10
; preds = %6
  store i32 1, i32* %1
  br label %11

; <label>:11
; preds = %10, %9, %5
  %12 = load i32* %1
  ret i32 %12
}
```

Figure 5. LLVM Bitcode of the `get_sign` function.

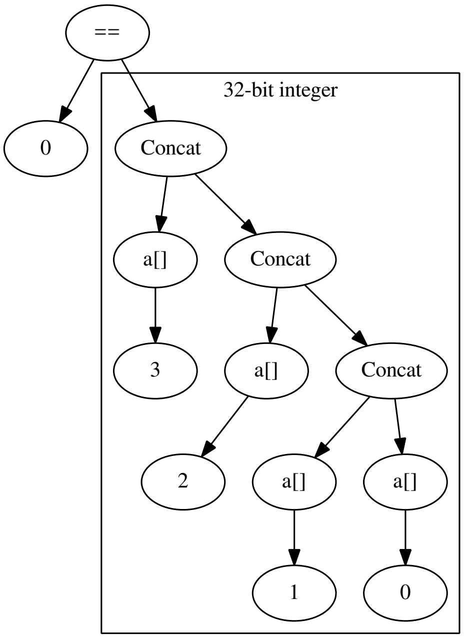


Figure 6. Flow graph for the path constraint `x == 0`.

stored as an array of bytes. Once the 32-bit value is formed, it is compared to zero.

When the first `if` statement is taken, the program eventually terminates with no more symbolic branches. A test case is produced: `x` = 0. Symbolic execution continues with the path constraint `x` $\neq$ 0 (so that the first `if` statement is not followed.) When the second `if` statement is encountered, the constraint `x` $<$ 0 is added to the path constraint. The flow graph representation of the full path constraint at the second if statement is shown in Figure 7. Klee's canonical form for expressions does not include a "not equal to" operator, so a comparison to false is used to create the `x` $\neq$ 0 expression. The second expression in the path constraint, `x` $<$ 0, is added to the graph, and the two expressions are conjuncted together (at the top node of the flow graph).

Again, execution eventually termites without any further branches depending on symbolic data. Another test case is output in which `x` is negative, and the expression `x` $<$ 0 is negated and added to the path constraint. The flow graph of the third and

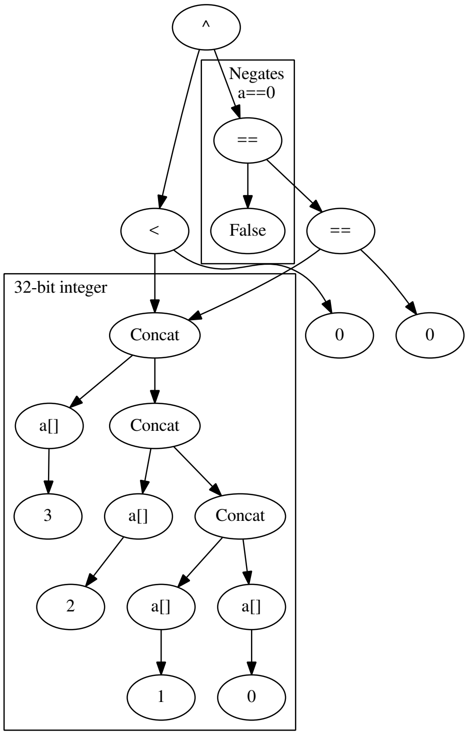


Figure 7. Flow graph for the path constraint `x != 0` and `x < 0`.

final expression sent to the solver is shown in Figure 8.

3.3.3 Hash Value Generation.

In order to quickly enumerate subgraphs, a spanning tree is found in the graph. A spanning tree in the flow graph of Figure 8 is shown in Figure 9. k-subtrees rooted at the root of the spanning tree are enumerated, and then the edges not contained within the spanning tree with both endpoints in the subtree are added in order to form a k-subgraph of the flowgraph.

Enumerating the 8-subgraphs of the flow graph in Figure 8 yields 78 subgraphs. Canonicalization shows that 15 of those subgraphs are unique. Each of these unique subgraphs is represented by a unique 64-character string of ones and zeros (the rows of the adjacency matrix concatenated together). The MD5 digest of each of those 15 strings is computed, and the 5 least digests are the min-hash signature for the

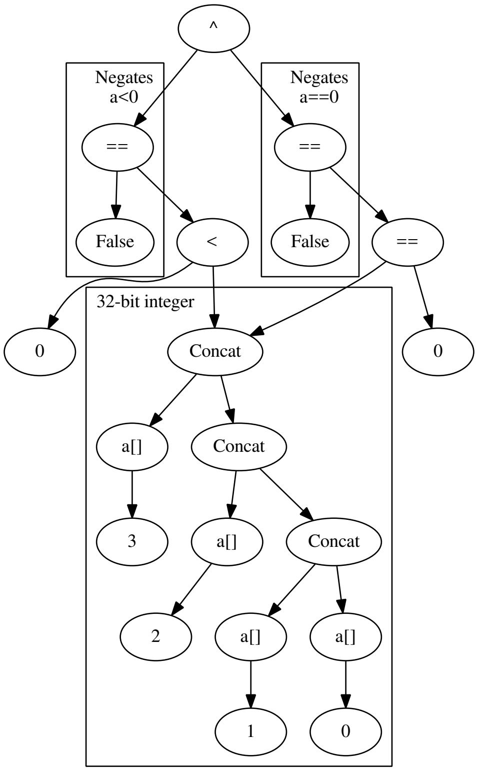


Figure 8. Flow graph for the path constraint `x != 0` and `x !< 0`.

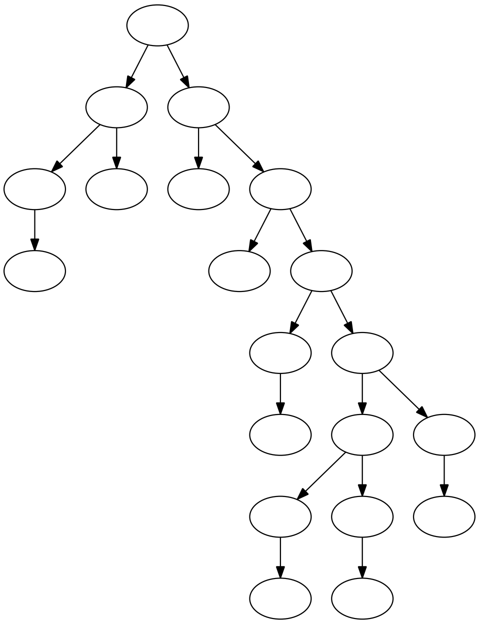


Figure 9. A spanning tree found within the flow graph of Figure 8.

path constraint. In this example, each of the three constraints shown has a different min-hash signature.

3.4 Implementation

TLSH was implemented in two versions of Klee [12], one that predates Klee's version numbering and includes Query Count Estimation (QCE) [36], and the most current version of Klee at the time of implementation (1.3.0). Klee provides the basic functionality required to merge two execution states together, this was augmented with TLSH, and a command-line option exposed to select whether to use TLSH. When merging two states, flow graph representations of their path constraints are formed using a custom graph class that makes use of STL containers. 8-subgraphs that include the root node enumerated, and passed to the Bliss [32] library for canonicalization using the default configuration. Min-hash[11] was then used to divide each set of queries into "buckets" of similar queries, using $n = 5$. Only states with constraints that hash to the same value are merged. TLSH hash values are "memoized", so if the path constraint is not modified between calls to the TLSH computation function, the TLSH is not re-computed.

3.5 Summary

The Tree Locality-Sensitive Hash (TLSH) method assigns a hash value to the path constraints solved during symbolic execution. The TLSH is generated by constructing a flow graph from the constraint, then finding particular k-subgraphs within the flow graph. Each subgraph is canonicalized, and min-hash is used to generate a hash value. TLSH with high probability assigns the same hash value to constraints that are structurally similar to each other, enabling the identification of similar constraints without performing a time-consuming pair-wise comparison between them.

IV. Results

Evaluation of the Tree Locality-Sensitive Hash (TLSH) method verifies that it both accurately identifies the nearest neighbor of each constraint, and that it is suitable for use in symbolic execution. All testing was performed using Klee [12] to symbolically execute the GNU Coreutils [2].

To verify the accuracy of the hash method, constraints sent to the SMT solver while executing the Coreutils were logged. TLSH was used to cluster the constraints, and the tree edit distance between some constraints was computed using Apted [52] in order to find the nearest neighbor of each constraint. The hash assigns the same hash value to a constraint and its nearest neighbor in excess of 99% of the time.

To examine the suitability of the hash for use in symbolic execution, TLSH was implemented in Klee, and a command-line option was added to select between different merging strategies. When combined with QCE [36], using the hashing method resulted in an increase in instructions symbolically executed per second and in states run until termination, without a significant decrease in the number of paths discovered versus using QCE alone.

4.1 Hash Accuracy

To verify that TLSH acts as an approximation of the tree edit distance between constraints, the pairwise distance between constraints was computed for select sets of constraints.

To generate the representative queries, the GNU Coreutils[2] version 6.11 were compiled into LLVM bitcode using Clang 3.4[37]. Each of the utilities tested in the Klee[12] paper was symbolically executed with query logging enabled. Two query logs were generated for each run, one that includes all queries and one that only includes queries that were actually sent to the solver.

The accuracy of the hash was evaluated by computing the tree edit distance between selected queries in order to find the nearest neighbor for each query. The TLSH values for each query were calculated, and the probability that the hash assigned the same signature to nearest neighbors was computed.

Since each query can be represented as a tree, the *tree edit distance* can be computed between two queries. Apted[51, 52] is a fast, portable library for computing the edit distance between two trees. Due to the number of pairwise comparisons required, the tree edit distance between each query in a set cannot be computed in a reasonable amount of time, so the distances between a sampling of queries was computed. Approximately 1000 comparisons were performed: 500 between queries with different TLSH values (queries in different clusters), and 500 between queries in with the same TLSH value (and thus in the same cluster). The average intra- and inter-cluster distance is shown in Figure 10.

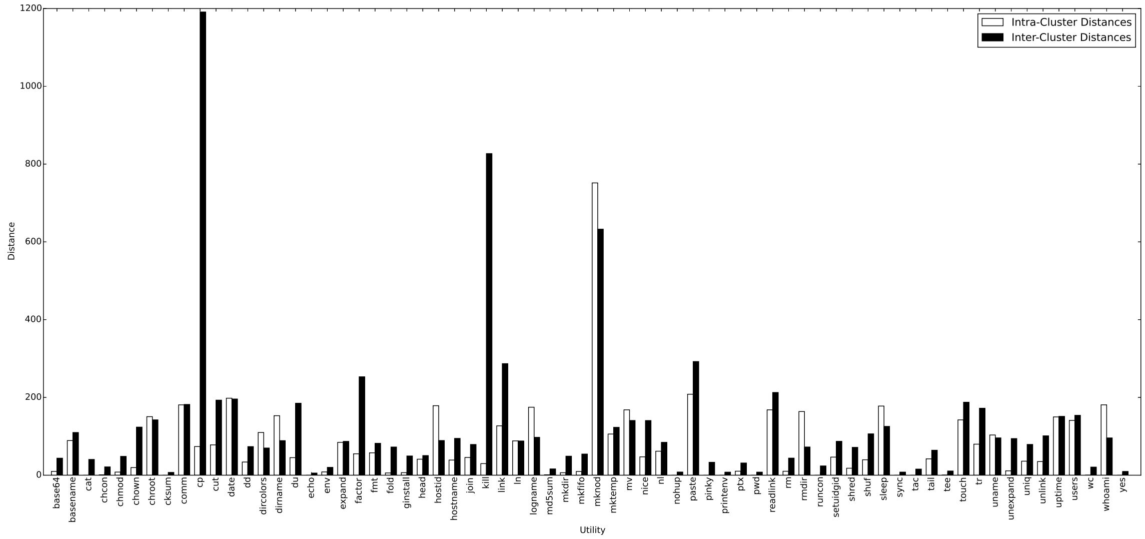


Figure 10. Average Tree Edit Distance Between 1000 SMT Constraints.

The results in Figure 10 do not conclusively show that the tree edit distance between queries in the same cluster is smaller on average than the tree edit distance between queries in different clusters. In order to better characterize the accuracy of the hash, five utilities were selected that captured the range of outcomes from the average tree edit distance analysis: cases where the intra-cluster distance was significantly less than the inter-cluster distance(`fold`), where the opposite was true(`hostid`, `logname`, `rmdir`), and where they were about the same(`date`). For each of these utilities, 1000 queries were chosen at random (with a uniform distribution), and the pairwise distances were computed between each query. Then, for each query that was selected the nearest neighbor or neighbors were found. If the query and at least one of its nearest neighbors were assigned the same signature by TLSH, the hash is considered to be accurate for that query. The results are shown in Table 1. In each case, the accuracy was above 99%.

Table 1. Hash Accuracy for Selected Utilities.

Utility	Accuracy
Date	99.9%
Fold	99.3%
Hostid	99.9%
Logname	99.8%
Rmdir	99.9%

This verifies that the TLSH is an accurate approximation of the tree edit distance between constraints, because it is very likely to assign the same hash value to a constraint and its nearest neighbor in a set for each of the five utilities tested. The accuracy and speed of TLSH indicate that it may be useful for selecting states to merge during symbolic execution; this is confirmed by implementing it in Klee and symbolically executing the Coreutils using state merging.

4.2 Use in Symbolic Execution

In order to compare the hash method with Query Count Estimation (QCE) [36], the hashing algorithm was implemented in a version of Klee that supports QCE. QCE was implemented by its authors in a variant of Klee in 2012, with the source code publicly released after publication. Since then, the Klee codebase has changed significantly: notably, the version of LLVM which Klee extends is updated from 2.9 to 3.4. Modern versions of Klee with QCE do not exist in any public repository, and the task of incorporating it is a software development task beyond the scope of this dissertation. Therefore, tests involving state merging were performed by incorporating TLSH in the publicly released version of Klee with QCE. In accordance with the recommendations of the Klee developers in 2012, the GNU Coreutils version 6.11 were compiled using `llvm-gcc`, a variant of GCC 4.2 that is capable of emitting LLVM bitcode.

Klee is based on LLVM, and borrows heavily from it's optimized data structures and algorithms, in addition to containing quite a bit of heavily optimized code of its own. The hashing algorithm was largely implemented using standard STL containers, and has not been optimized beyond memoization of hash values for execution states.

4.2.1 Test Setup.

The previously published best strategy for merging states is to perform a topological search of the control-flow graph (CFG), attempting to merge states at control flow join points. Whether two states are merged or not depends on the Query Count Estimate (QCE) [36] heuristic, which is an estimate of the increase or decrease in calls to a SMT solver if two states are merged or if each state continues execution independently. One weakness of QCE that was identified by the authors is that it does not take into account the difficulty of future solver calls, just the number of calls.

Since QCE doesn't take into account the difficulty of future solver calls, the minhashing approach is used along or in combination with QCE to further restrict the merging of states so that states with dissimilar path constraints are not merged.

To provide a baseline measure of the difficulty of symbolically executing a particular program and set of inputs, each program is symbolically executed without merging any states, selecting states at random for further execution. Table 2 provides an overview of the strategies tested.

Table 2. Merging Strategies Tested.

Strategy	Description
Baseline	No Merging; random search
Naive	All possible merges; topographical search
QCE	Merge based on QCE; topographical search
TLSH	Merge states with same hash; topographical search
Combined TLSH & QCE	Merge based on QCE, but only states with the same hash; topographical search

The authors of Klee have published a set of symbolic input configurations that achieve high test coverage on the Coreutils suite. The symbolic inputs differ significantly from those used to test QCE [36]. When testing QCE, many (dozens) of short (two-byte) symbolic command-line arguments were used to test each tool. The original Coreutils tests using Klee utilize many fewer symbolic arguments that are longer, in addition to symbolic files in some cases. In addition to providing high code coverage, Klee found several previously unknown bugs while using these inputs. Therefore, while some metrics (i.e. number of paths executed) may be improved by using many short symbolic arguments, the configuration published by Klee's authors is closer to how software would be tested in the "real world." This symbolic input configuration was used, and the Coreutils were symbolically executed for up to an hour each using each of the merging strategies discussed above. More information on how the choice of symbolic inputs affects performance is available in Appendix B.

Three different measures are recorded and analyzed: instructions executed, states executed to completion, and partial paths discovered. The instructions executed is the number of LLVM [37] instructions executed by the Klee virtual machine. This number is reported by the virtual machine, and is a measure of how many times the symbolic execution loop in Figure 1 was traversed. States executed to completion is the number of execution states that halt, either due to Klee finding a bug or reaching a program exit point. A counter is incremented when a halting state is discovered, and this is when Klee generates a concrete test case that causes the program to traverse the path traveled by the halting state. The partial paths discovered is an estimate of the number of paths that have been analyzed thus far by Klee. The estimate is based on the combined *multiplicity*, which is a measure of how many states there are currently and how many states were merged to create the current states [36]. This includes both paths that have halted and those that have not halted yet (paths explored by execution states that are still in the state pool, requiring further execution).

4.2.2 Instructions Executed in 60 Minutes.

Figure 11 shows the number of instructions executed during 60 minutes using the minhash merging strategy and using a naive merging strategy that merges states whenever possible.

Results are presented on log-log graphs, with a the line $y = x$ plotted as a solid line. Points to the right of the line represent utilities where there was a gain in performance when using the configuration on the y-axis, in this case Minhash. The majority of the points lie to the right of the line, indicating that the Minhash merging strategy does successfully mitigate against creating a state with a more complex path constraint.

QCE attempts to reduce the number of queries that are sent to the SMT solver,

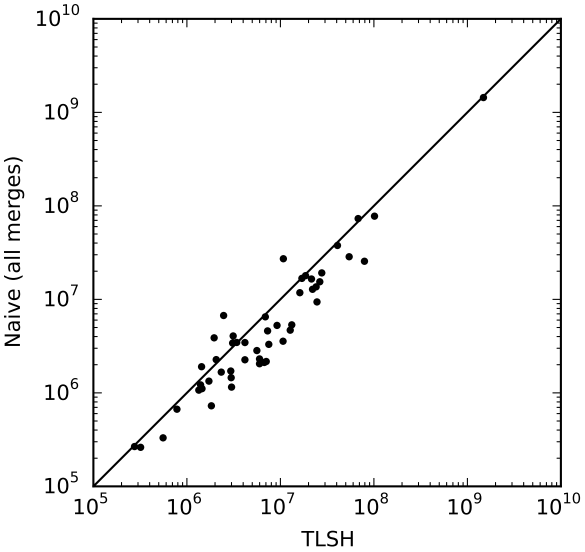


Figure 11. Instructions Executed in 60 Minutes Using Naive Merging vs TLSH

which should also increase the rate of symbolic execution. Figure 12 compares the instructions executed under the minhash strategy compared to a combined strategy that uses both QCE and minhash, and Figure 13 compares the combined strategy to QCE alone. The combined merging strategy executes more instructions than either merging strategy alone.

The combined merging strategy executes more instructions than any other strategy that merges states together. Still, even a perfect merging strategy should cause some overhead, because merging states together takes processor time that could be spent executing instructions. Figure 14 compares the combined merging strategy to symbolic execution with merging disabled.

There are several utilities where the baseline strategy (no merging) executed more instructions than the combined merging strategy, but in the majority of cases, more instructions were executed with state merging than without it. This is a somewhat counter-intuitive result; the path constraints of the unmerged states should be simpler to solve than those of merged states, and there is some overhead associated with

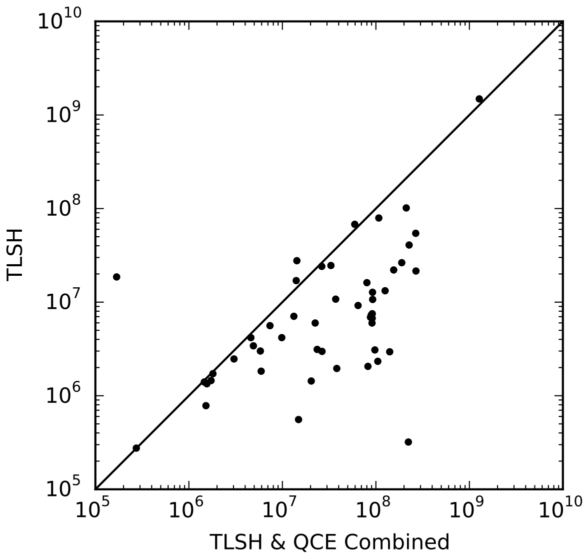


Figure 12. Instructions Executed using TLSH vs TLSH & QCE Combined.

selecting states for merge and then actually merging them together. Figure 15 shows how the number of instructions executed grows over time for the utility `nice` for each strategy.

When symbolic execution first begins, the results are as expected: disabling merging increases the instructions per second. At a certain point, however, the system essentially stops executing instructions. This seems to happen when Klee consumes so much RAM that the system starts paging (the system had 32 gigabytes of RAM for these tests, and no other applications were open during the test except for a terminal window to monitor progress.) The only solution available in Klee to mitigate this problem is to start discarding states when a memory limit is reached, but this function has a ceiling of one gigabyte in the version of Klee that contains QCE, which results in a significant number of states being discarded. Every discarded state is a number of lost opportunities to find bugs and enhance coverage.

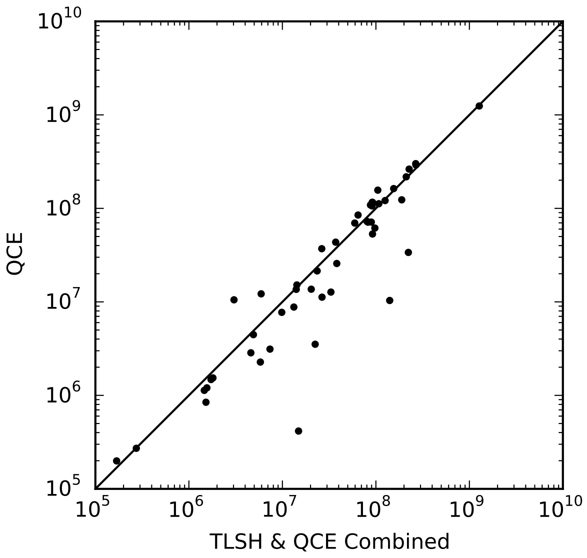


Figure 13. Instructions Executed using QCE vs TLSH & QCE Combined.

4.2.3 States Executed to Completion in 60 Minutes.

When a halting condition is reached, Klee makes a final series of queries to the SMT solver to generate a concrete test case that follows the same path through the CFG that the halting execution state traversed. At every opportunity, Klee checks the execution state to attempt to expose a bug, such as illegal memory access, and failed to find an error. Thus, the more execution states that terminate, the more faith we have in our coverage. This test counts the number of execution states that terminate in 60 minutes.

In most cases, Klee did not output any terminated execution states without using merging. Figure 16 shows the number of states executed to completion by the combined merging strategy and without merging. The diamonds on the x-axis of Figure 16 represent utilities which completed a number of paths when merging was enabled but did not complete any without state merging. Figures 17, 18, and 19 show the number of states executed to completion by the combined merge strategy compared to a naive strategy, minhash alone, and QCE alone respectively.

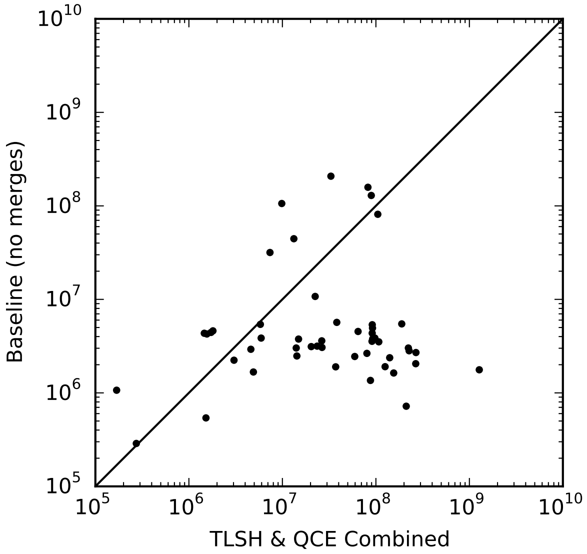


Figure 14. Instructions Executed using Baseline (No Merging) vs TLSH & QCE Combined.

The combined merging strategy that uses both QCE and TLSH to limit which states are merged finds more terminal execution states, on average, than any other merging configuration tested. This means that more test cases can be generated, and we have a higher confidence that there are no undiscovered errors lurking in the program being tested.

4.2.4 Partial Paths Discovered.

When published, QCE was evaluated largely based on the estimated number of partial paths explored during a fixed time interval. Counting the paths explored with state merging disabled is easy: we can count the number of feasible branches we have taken. Directly counting the number of paths explored when states have been merged is more difficult.

The *multiplicity* of an execution state is a measure of how many paths lead to that execution state. When symbolic execution begins, it begins with a single execution state of multiplicity one. When a branching instruction is encountered, the multipli-

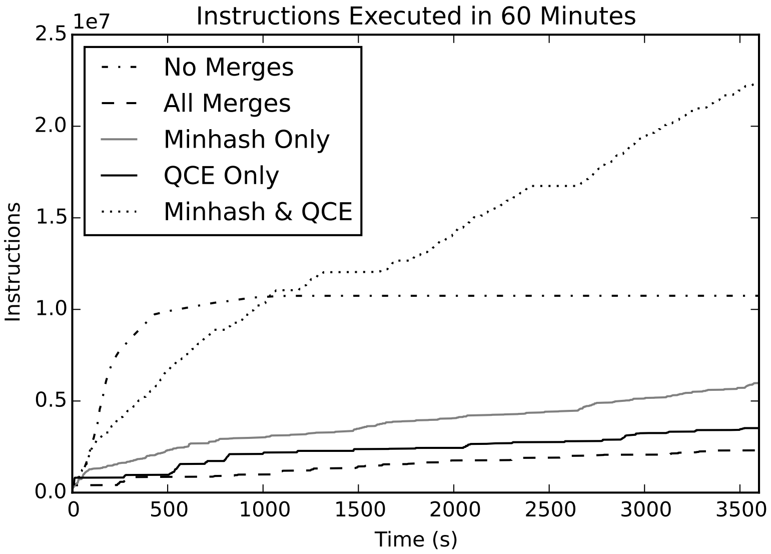


Figure 15. Instructions Executed in `nice`.

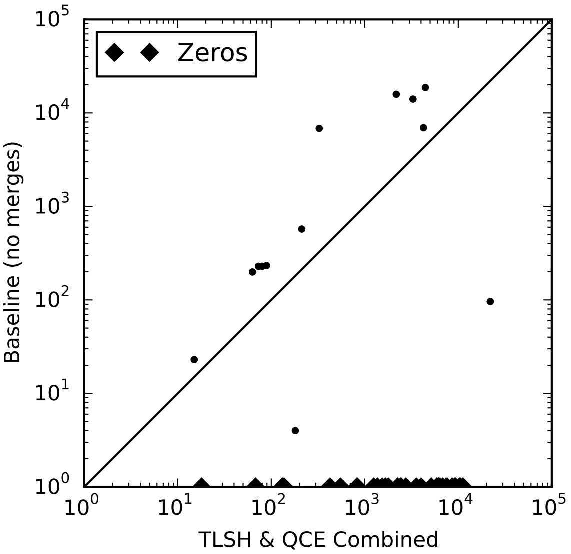


Figure 16. Terminating States using Baseline (No Merging) vs TLSH & QCE Combined.

city of the resulting states is the same as the multiplicity of the state that reached the branching instruction. When two states are merged, the multiplicity of the new state is the sum of the multiplicities of the merged states. State multiplicity overestimates the number of feasible paths that have been executed, because it assumes that if a

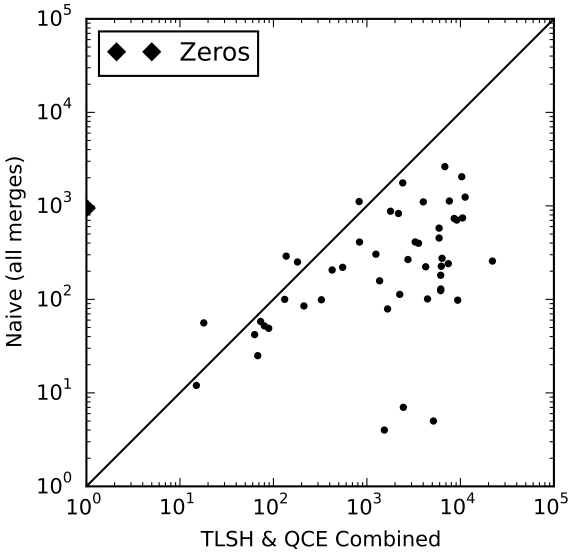


Figure 17. Terminating States using Naive Merging (All Merges) vs TLSH & QCE Combined.

branch is feasible for particular state, then it would be feasible for every state that was merged to form that state.

In order to estimate the number of paths discovered using the multiplicity of execution states, the path explosion problem is modeled as $a \cdot 2^{b \cdot n}$ where a and b are program-specific constants, and n is the growth parameter that indicates the progress made in symbolically executing the program thus far[36]. Both the path count p and the state multiplicity m can be modeled this way, with different values of a and b. The growth parameter n is the same between them, as it depends on the search strategy. Thus, $m = a_m \cdot 2^{b_m \cdot n}$ and $p = a_p \cdot 2^{b_p \cdot n}$. Solving for n and substituting it yields $\log p \approx c_1 + c_2 \log m$.

This relationship can be validated empirically by manually keeping track of the states that are merged to keep an accurate record of path count and state multiplicity. This record keeping approach also allows for estimates of c_1 and c_2. A least-squares linear regression can be used to estimate c_1 and c_2 for a given program. Note that this process must be repeated for each program and search strategy, doubling the

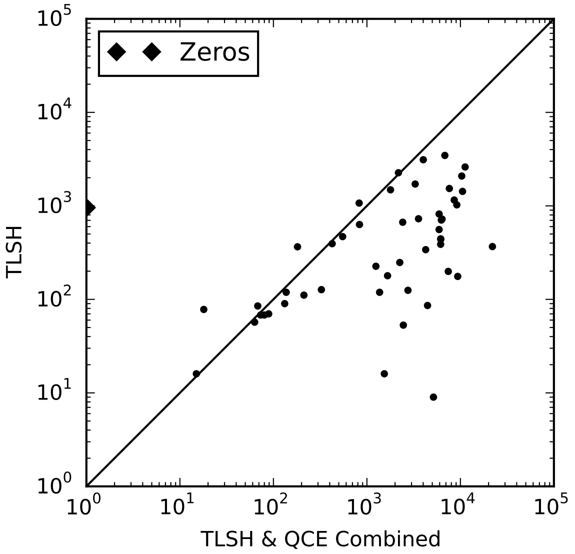


Figure 18. Terminating States using TLSH vs TLSH & QCE Combined.

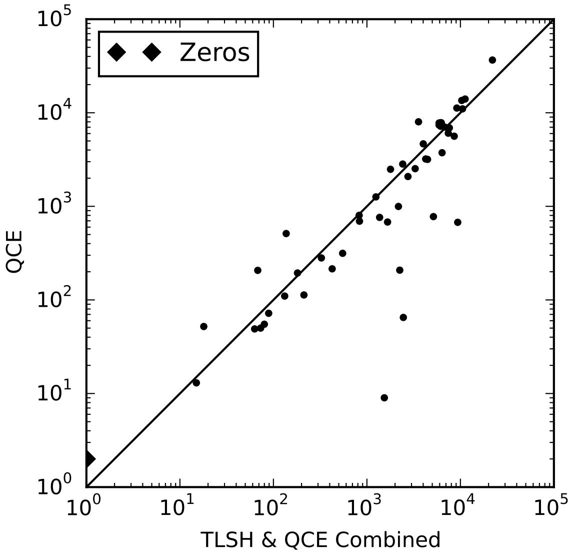


Figure 19. Terminating States using QCE vs TLSH & QCE Combined.

time necessary to perform each experiment.

Figure 20 shows the estimated number of partial paths explored in 60 minutes using the combined merging strategy and QCE alone.

Most of the cases are around the unity line, but four utilities exhibited a significant

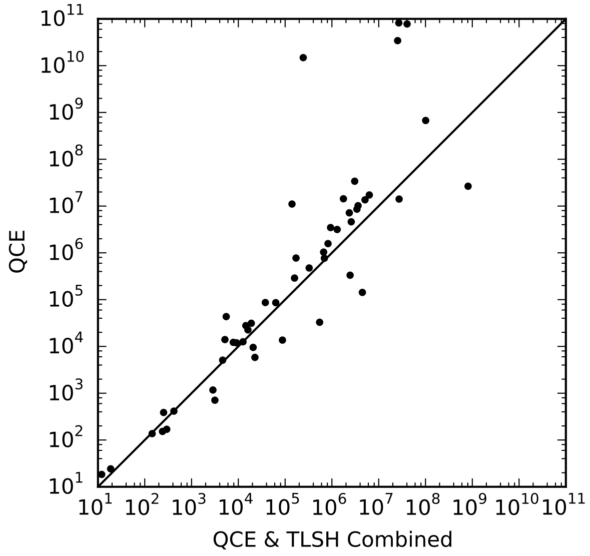


Figure 20. Estimated Paths Explored using QCE vs TLSH & QCE Combined.

increase in paths discovered using QCE alone: `chcon`, `shred`, `cat`, and `tac`. `Chcon`, `shred`, and `cat` spend more than two-thirds of the hour attempting to merge states, so something about the constraints being generated causes the hashing operation to be more expensive in these cases. Unlike the rest of Klee, implementation of the hashing algorithm uses standard STL containers, and could probably be optimized further to reduce the overhead associated with it. Alternatively, it is possible to recognize dynamically that Klee is spending a large percentage of its time calculating the TLSH values, and TLSH could be disabled automatically. `Tac` didn't exhibit the same slowness as the other three utilities, but using TLSH in combination with QCE didn't produce gains over using QCE alone.

4.3 Summary

The approximate structural hash assigns the same hash value to constraints that are close to each other under the tree edit distance metric. When a combined strategy of QCE and constraint hashing is used to select states for merging, more instructions per second are executed than with any other merge strategy. This causes more execution states to be executed to completion as well. QCE alone discovers slightly more partial paths (partial in the sense that they do not halt), but this is most likely due to the overhead associated with actually computing the hash, not because QCE alone is a superior strategy for selecting states.

V. Conclusion

Symbolic execution allows software testers to make certain inputs symbolic, and then explore every feasible path through the control-flow graph (CFG), checking for errors as each instruction is executed, and finally generating a concrete test case that causes execution to follow that path. The number of paths grows exponentially with the number of branches in the CFG, and determining the feasibility of the graph grows exponentially more difficult as the branch condition depends on more of the symbolic input. These two exponentially growing problems, respectively known as the path explosion problem and state explosion problem, cause symbolic execution to consume significant amount of time and memory for any configuration of “real-world” programs and inputs.

5.1 Summary

Previous work provides a method for minimizing the increase in the number of queries from merging states together. This work attempts to avoid creating performing merges that result in more difficult queries by examining the structure of the path constraints of execution states to be merged, and only allowing the merge if the path constraints are similar.

Similarity is evaluated by a novel Tree Locality-Sensitive Hash (TLSH) algorith that represents the path constraint of each execution state as a directed acyclic graph (DAG), finds certain subgraphs of size k in that graph, and uses those subgraphs as shingles to compute a minhash hash of the graph. States are only merged if their path constraints have the same TLSH hash.

When applied to path constraints solved during symbolic execution, TLSH accurately assigns the same hash value to constraints that are nearest neighbors under

Tree Edit Distance. Combining Query Count Estimateion (QCE) [36] and TLSH increased the number of instructions executed per second over any other merge strategy tested. It also allowed Klee to execute more execution states until they halted, allowing for the creation of more concrete test cases. QCE alone discovered slightly more partial paths overall, likely due to the overhead of computing the graph hashes. The graph hashing code could likely be tuned to provide greater performance.

5.2 Future Work

The state merging results presented here use a significantly outdated version of Klee, because that is the only version that includes QCE. Newer versions of Klee offer features that are a significant improvement: more diverse state selection strategies, support for more than 1GB of RAM, more control of symbolic inputs, and most significantly it is based on a newer version of LLVM.

QCE and structural constraint hashing should be ported forward to mainline Klee, which is still being actively developed. The results presented in Appendix A indicate that state merging would also be a viable strategy to mitigate the path explosion problem using newer versions of Klee, although preliminary testing indicates that there may be performance bottlenecks that occur during the memory-map comparison that occurs while merging states, but these cannot be precisely identified until the merging algorithms are ported forward.

Since no one merging strategy wins in every case, a strategy that was able to dynamically select between merging strategies during symbolic execution would likely be able to achieve better results than any one strategy alone. While there is likely no way to know *a priori* which strategy will be best for a given program and set of symbolic inputs, there is enough information available during symbolic execution to identify likely missteps, and to correct them.

Newer versions of Klee also support additional SMT solvers, notably Z3. It is possible that the performance of Z3 would significantly differ from that of STP, which is the solver used when merging states here.

Appendix A: State Merging Simulation

To verify that merging states selected by the approximate structural hash algorithm does not create a path constraint that is significantly harder to solve after the merge, a simulation of the operations performed during state merging was performed.

The query replay tool included with Klee, `kleaver`, was modified to solve the buckets by first disjuncting the queries in a bucket together, and then solving the resulting query. This simulates part of what occurs when two execution states are merged during symbolic execution. The time required to solve each of the queries individually and to solve the entire disjuction was recorded using the Linux `time` tool. The average speedup over all the buckets in each utility is shown in Figure 21. A speedup of 0 indicates that it took the same amount of time to solve the queries in a bucket sequentially as it did to disjunct all of them together and solve the result, positive numbers indicate that it was faster to solve the disjunction than solve the queries sequentially.

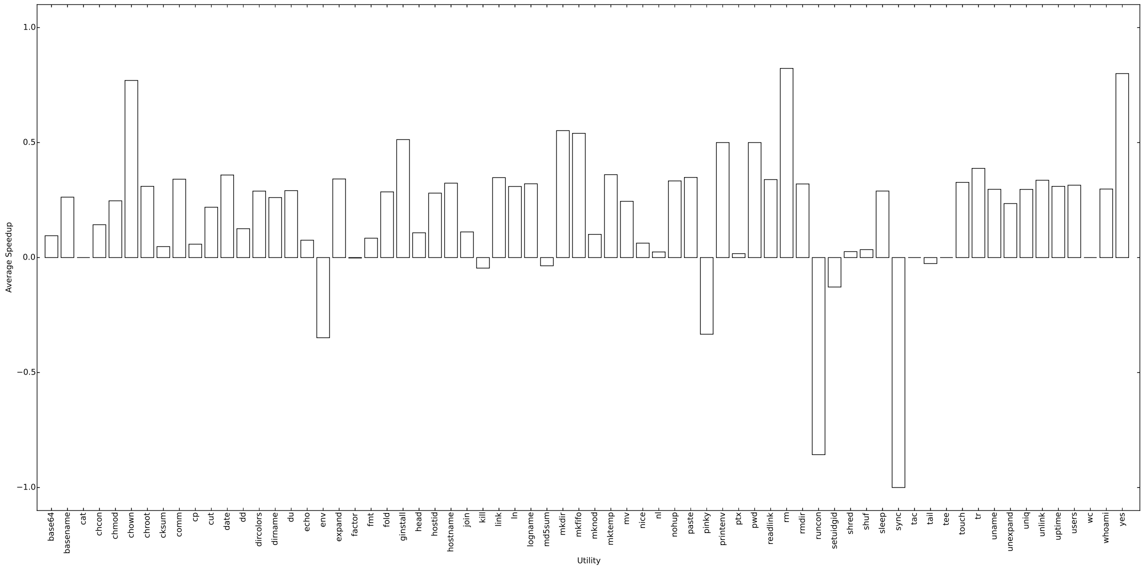


Figure 21. Average Speedup for each Coreutil.

Of the 70 utilities tested, 55 of them (70%) experienced a speedup. Of the cases where disjuncting the queries was faster than solving them sequentially, the average speedup was 29%. For a few of the utilities, notably seq, it was significantly slower to disjunct the queries together and solve them. The seq utility accepts three parameters as input: a starting value, an ending value, and a step size. It then prints a sequence of numbers from the start value to the end value with the indicated step size. In order to maximize the precision on a variety of different platforms, the values are read out of the string and converted to a custom data structure that utilizes a long double data type. This custom data structure is likely what causes the disjunction to be so bad. The STP [23] solver does not handle disjunctions involving arrays well, because the solving loop ends up running many times. Solving the same disjunctions in Z3 [9] is slower than solving the queries sequentially, but the slowdown is not nearly as pronounced. The results from that test are summarized in Table 3. More data from the symbolic execution and state merging simulation can be found in the Appendix.

Table 3. Comparison between Z3 and STP for Seq and Pathchk.

Utility	Speedup (STP)	Speedup (Z3)
pathchk	-5.14	-2.11
seq	-18.8	-0.686

This test shows that for most of the applications tested, it is faster to merge the path constraints of states selected by the approximate structural hash than it is to solve each of the constraints separately. This should translate into faster test case generation when state merging is used to reduce the number of paths that are symbolically executed.

Appendix B: Selecting Symbolic Input

The choice of what inputs to make symbolic, along with their maximum size, has a dramatic effect on the behavior of the symbolic execution algorithm. Small symbolic inputs limit the range of the program that can be reached, while large symbolic inputs can result in path constraints that quickly overwhelm the SMT solver and limiting the execution depth that can be achieved in a reasonable amount of time.

Symbolic inputs available in Klee [12] include symbolic files, symbolic command line arguments, and a symbolic *stdin*. Selecting appropriate symbolic input for a program must be done on a case-by-case basis, as each program is designed to handle input differently. As an example, consider the utility `link`, part of the Coreutils suite. According to its documentation, `link` has two valid invocations: `link FILE1 FILE2`, which creates a link named `FILE2` to an existing file named `FILE1`, and `link OPTION`, where `OPTION` is either `--help` or `--version`, which causes link to either print its documentation or version, respectively. Given the task of creating inputs that might cause `link` to fail in some way, a number of scenarios may immediately present themselves: what if `link` is called with no arguments, or with more than two arguments? What if `FILE1` doesn't exist, or what if there is already a file named `FILE2`? What if the option supplied to link is longer than either expected option? With proper configuration of the symbolic inputs, the behavior of `link` under each of these scenarios can be explored.

The running times and memory requirements of search algorithms are evaluated based on their growth for some size parameter n. Because of the complexity of symbolic execution, different symbolic search spaces of the same size will yield dramatically different results, even when testing the same program. Previous research identified the coreutil `nice` as a utility that experiences about an average speedup when state merging is enabled using Query Count Estimation as a heuristic to merge states. Two experiments were set up: executing `nice` with n 2-byte symbolic arguments, and with 2 n-byte arguments. In both case, there are $2n$ bytes of symbolic search space, and when $n = 2$ the two experiments are the same. With n larger than 2, the second experiment (2 longer arguments) times out, failing to explore every path through the program in one hour. The running times for the experiment with n 2-byte arguments can be seen in Figure 22.

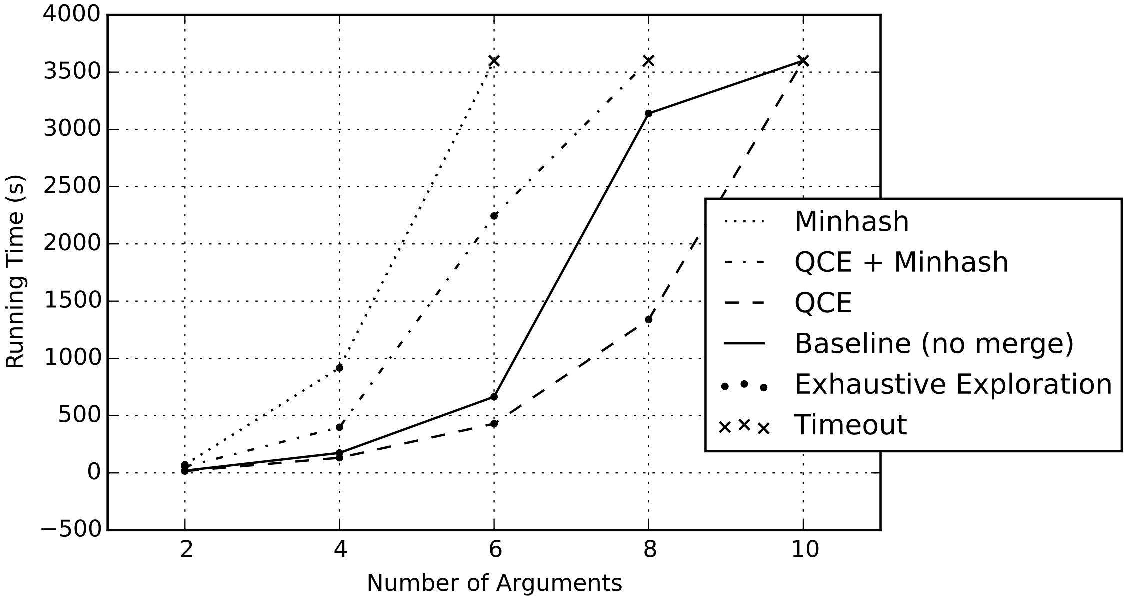


Figure 22. Running Time to Execute `nice` with 2-byte Arguments.

Regardless of the merging strategy, the running time grows roughly exponentially with the size of the input space, until eventually timing out for each configuration. The QCE strategy alone shows the slowest growth, and is the only strategy to improve upon the baseline search (random state selection without merging) in any of the cases.

Since any configuration with arguments that are more than two bytes fails to complete in an hour, one way to compare them is by estimating the number of distinct paths that have been explored thus far. Figure 23 shows the number of paths executed in one hour for several different input sizes and merging strategies. Note that the yaxis is logarithmic.

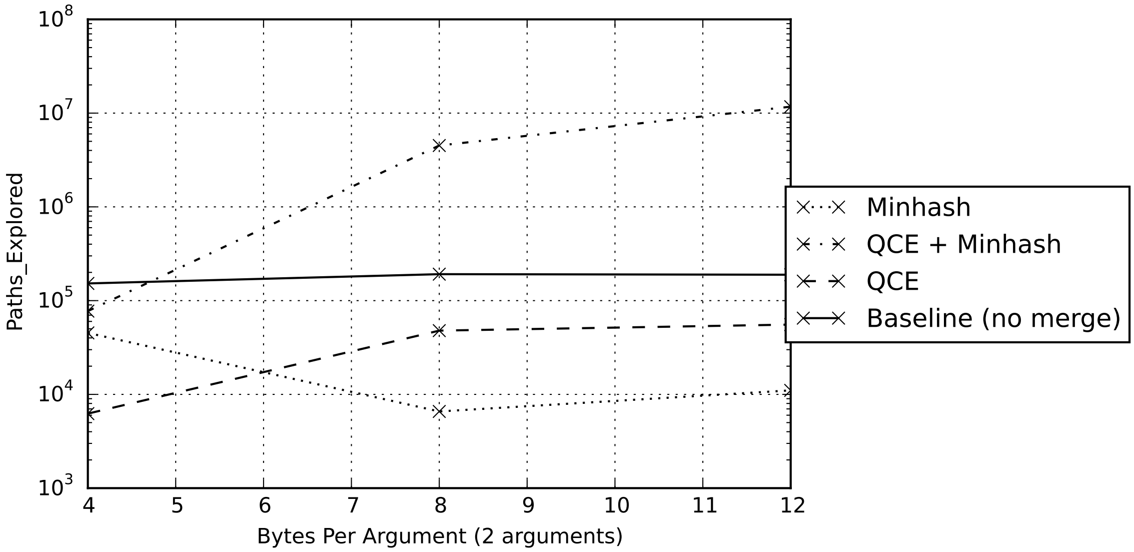


Figure 23. Paths Explored in 60 Minutes.

As the size of the symbolic input space increases, the number of paths explored in one hour does not increase exponentially, because the amount of computational effort that must be expended to explore a path does not decrease. The fact that fewer paths are explored with smaller inputs is likely explained by the fact that symbolic execution reaches deeper in the control flow graph, so that the complexity of each execution state increases. When the symbolic arguments are 8 or 12 bytes long, the only merging strategy that increases the number of paths explored over the baseline configuration is a combination of QCE and constraint hashing.

Bibliography

1. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl. August 2017.

2. Coreutils. http://www.gnu.org/software/coreutils. January 2016.

3. Pathgrind. https://github.com/codelion/pathgrind. July 2016.

4. Christopher G. Atkeson, Andrew W. Moore, and Stefan Schaal. Locally Weighted Learning. *Artificial Intelligence Review*, 11:11–73, 1997.

5. Domagoj Babi and Alan J. Hu. Approximating the safely reusable set of learned facts. *International Journal on Software Tools for Technology Transfer*, 11(4):325–338, 2009.

6. Clark Barrett and Sergey Berezin. CVC Lite: A New Implementation of the Cooperating Validity Checker. In Rajeev Alur and Doron A Peled, editors, *Proceedings of the 16th International Conference on Computer Aided Verification (CAV '04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518. Springer-Verlag, 2004.

7. Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. A taint based approach for smart fuzzing. *Proceedings - IEEE 5th International Conference on Software Testing, Verification and Validation, ICST 2012*, pages 818–825, 2012.

8. Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. Framework for instruction-level tracing and analysis of program executions. *Proceedings of the 2nd international conference on Virtual execution environments - VEE '06*, page 154, 2006.

9. Nikolaj Bjorner and Leonardo de Moura. Z3: An efficient SMT solver. *Lecture Notes in Computer Science*, 4963:337–340, 2007.

10. Peter Boonstoppel, Cristian Cadar, and Dawson Engler. RWset: Attacking Path Explosion in Constraint-Based Test Generation. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 351–366, 2008.

11. Andrei Z. Broder. On the Resemblance and Containment of Documents. In *Proceedings of the Compression and Complexity of Sequences 1997*, pages 21–29, 1997.

12. Christian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. *USENIX Symposium on Operating Systems Design and Implementation*, pages 209–224, 2008.

13. Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: Automatically Generating Inputs of Death. *Proceedings of the 13th ACM conference on Computer and communications security - CCS '06*, 12(2):322, 2006.

14. Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Pasareanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: preliminary assessment. *2011 33rd International Conference on Software Engineering (ICSE)*, pages 1066–1071, 2011.

15. Silvio Cesare and Yang Xiang. Malware variant detection using similarity search over sets of control flow graphs. *Trust, Security and Privacy in Computing and Communications*, pages 181–189, 2011.

16. Christian Collberg. *Surreptitious Software*. Addison-Wesley Professional, 2009.

17. Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd IEEE Symposium on the Theory of Computation*, pages 151–158, 1971.

18. Patrick Copeland. *Using State Merging and State Pruning to Address the Path Explosion Problem Faced by Symbolic Execution.* Master's thesis, Air Force Institute of Technology, 2014.

19. Derek G. Corneil and Calvin C. Gotlieb. An Efficient Algorithm for Graph Isomorphism. *Journal of the ACM*, 17(1):51–64, 1970.

20. Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM ACM*, 5(7):394–397, 1962.

21. Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201–215, 1960.

22. Leonardo de Moura and Nikolaj Bjørner. Model-based Theory Combination. *Electronic Notes in Theoretical Computer Science*, 198(2):37–49, 2008.

23. Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. *Internation Conference on Computer Aided Verification*, pages 524–536, 2007.

24. Patrice Godefroid. Compositional Dynamic Test Generation. *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 47–54, 2007.

25. Patrice Godefroid, Michael Y. Levin, and David Molnar. SAGE: Whitebox Fuzzing for Security Testing. *Queue*, 10(1):40–44, 2012.

26. Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated Whitebox Fuzz Testing. *Network and Distributed System Security Symposium (NDSS)*, 9(July):155–166, 2008.

27. John B. Goodenough and Susan L Gerhart. Toward a theory of test data selection. *ACM SIGPLAN Notices*, 10(6):493–510, 1975.

28. Trevor Hansen, Peter Schachte, and Harald Søndergaard. State Joining and Spliting for the Symbolic Execution of Binaries. In *9th International Workshop on Runtime Verification, RV 2009*, pages 76–92, 2009.

29. Stephen G. Hartke and A.J. Radcliffe. McKay's Canonical Graph Labeling Algorithm. *Contemporary Mathematics*, pages 99–112, 2007.

30. Piotr Indyk and Rajeev Motwd. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proceedings of the Thirtieth Annual ACM symposium on Theory of Computing*, pages 604–613, 1998.

31. Wesley Jin, Sagar Chaki, Cory Cohen, Arie Gurfinkel, Jeffrey Havrilla, Charles Hines, and Priya Narasimhan. Binary function clustering using semantic hashes. *Proceedings - 2012 11th International Conference on Machine Learning and Applications, ICMLA 2012*, pages 386–391, 2012.

32. Tommi Junttila and Petteri Kaski. Engineering an Efficient Canonical Labeling Tool for Large and Sparse Graphs. *In Ninth Workshop on Algorithm Engineering and Experiments*, pages 135–149, 2007.

33. William B. Kimball. *A Formal Approach to Vulnerability Discovery in Binary Programs.* Ph.D. Dissertation, Air Force Institute of Technology, 2013.

34. Jesse Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation*, 3:91–97, 2006.

35. Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Polymorphic worm detection using structural information of executables. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 3858 LNCS, pages 207–226, 2006.

36. Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '12*, pages 193–204, 2012.

37. Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, CGO*, pages 75–86, 2004.

38. Jure Leskovek, Anand Rajaraman, and Jeffrey D Ullman. *Mining of Massive Datasets.* Cambridge University Press, 2010.

39. You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. Steering symbolic execution to less traveled paths. *ACM SIGPLAN Notices*, 48(10):19–32, 2013.

40. Rupak Majumdar and Koushik Sen. Hybrid concolic testing. *Proceedings - International Conference on Software Engineering*, pages 416–425, 2007.

41. Rupak Majumdar and Koushik Sen. Latest: Lazy dynamic test input generation. Technical report, 2007.

42. Emaad A. Manzoor, Sadegh Momeni, Venkat N. Venkatakrishnan, and Leman Akoglu. Fast Memory-efficient Anomaly Detection in Streaming Heterogeneous Graphs. *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, page 10, 2016.

43. Robert Cunningham Michael Zhivich. The Real Cost of Software Defects. *IEEE Security & Privacy Magazine*, 7(2):87–90, 2009.

44. Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, 1990.

45. Glenford J Myers, Todd M Thomas, and Corey Sandler. *The Art of Software Testing 3rd Edition*, volume 1. 2011.

46. Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. Automatically Classifying Benign and Harmful Data Races Using Replay Analysis. *ACM SIGPLAN Notices*, 42(6):22–31, 2007.

47. Greg Nelson and Derek C. Oppen. Simplification by Cooperating Decision Procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):245–257, 1979.

48. Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2):47–69, 2003.

49. Robert Nieuwenhuis and Albert Oliveras. On SAT Modulo Theories and Optimization Problems. *Theory and Applications of Satisfiability Testing - SAT 2006*, 4121(i):156–169, 2006.

50. Mateusz Pawlik and Nikolaus Augsten. RTED : A Robust Algorithm for the Tree Edit Distance. *The 38th International Conference on Very Large Data Bases*, pages 334–345, 2011.

51. Mateusz Pawlik and Nikolaus Augsten. Efficient Computation of the Tree Edit Distance. *ACM Transactions on Database Systems (TODS)*, 40(3):1–40, 2015.

52. Mateusz Pawlik and Nikolaus Augsten. Tree edit distance: Robust and memory-efficient. *Information Systems*, 56:157–173, 2016.

53. William W. Peterson and Daniel T. Brown. Cyclic Codes for Error Detection. *Proceedings of the Institute of Radio Engineers*, 49(1):228–235, 1961.

54. Liva Ralaivola, Sanjay J. Swamidass, Hiroto Saigo, and Pierre Baldi. Graph Kernels for Chemical Informatics. *Neural Networks*, 18(8):1093–1110, 2005.

55. Charles Rich and Linda M. Wills. Recognizing a Program's Design: A Graph-Parsing Approach. *IEEE Software*, 7(1):82–89, 1990.

56. Ronald L. Rivest. The MD5 message-digest algorithm, 1992.

57. Koushik Sen. DART: Directed Automated Random Testing. *Hardware and Software: Verification and Testing*, 6405:213–223, 2011.

58. Koushik Sen, Darko Marinov, Gul Agha, Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A Concolic Unit Testing Engine for C. *10th European Software Engineering Conference and 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'05)*, 30(5):263, 2005.

59. Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. (State of) The Art of War: Offensive Techniques in Binary Analysis. *2016 IEEE Symposium on Security and Privacy (S&P)*, pages 138–157, 2016.

60. Chc Teixeira, Arlei Silva, and Wagner Meira Jr. Min-Hash Fingerprints for Graph Kernels: A Trade-off among Accuracy, Efficiency, and Compression. *Journal of Information and Data Management*, 3(3):227–242, 2012.

61. Nikolai Tillmann and Jonathan de Halleux. Pex: White Box Test Generation for .NET. *Tests and Proofs*, pages 134–153, 2008.

62. Nicholas Wells. Busybox: A swiss army knife for linux. *Linux Journal*, 2000(78es):10, 2000.

63. Elaine J. Weyuker and Thomas J. Ostrand. Theories of Program Testing and the Application of Revealing Subdomains. *IEEE Transactions on Software Engineering*, 6(3), 1980.

64. Laura A. Zager and George C. Verghese. Graph similarity scoring and matching. *Applied Mathematics Letters*, 21(1):86–94, 2008.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704–0188*

1. REPORT DATE *(DD–MM–YYYY)* 14–09–2017	**2. REPORT TYPE** Ph.D Dissertation	**3. DATES COVERED** *(From — To)* Aug 2008 — Sept 2017

4. TITLE AND SUBTITLE A Tree Locality-Sensitive Hash for Secure Software Testing	**5a. CONTRACT NUMBER** **5b. GRANT NUMBER** **5c. PROGRAM ELEMENT NUMBER**
6. AUTHOR(S) Camdon R. Cady, Captain, USAF	**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-DS-17-S-005
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Department of Electical and Computer Engineering 2950 Hobson Way WPAFB OH 45433-7765 DSN 785-2024, COMM 937-255-2024	**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 material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT

In order to eliminate bugs, developers may use symbolic execution to search through possible program states looking for anomalous states. Most of the computational effort to search through these states is spent solving path constraints in order to determine the feasibility of entering each state. State merging can make this search more efficient by combining program states, allowing multiple execution paths to be analyzed at the same time. However, a merge with dissimilar path constraints dramatically increases the time necessary to solve the path constraint. Currently, there are no distance measures for path constraints, and pairwise comparison of program states is not scalable. A hashing method is presented that clusters constraints in such a way that similar constraints are placed in the same cluster without requiring pair-wise comparisons between queries. When combined with other state-of-the-art state merging techniques, the hashing method allows the symbolic executor to execute more instructions per second and find more terminal execution states than the other techniques alone, without decreasing the high path coverage achieved by merging many states together.

15. SUBJECT TERMS

Software Testing, Symbolic Execution

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, AFIT/ENG
U	U	U	U	78	**19b. TELEPHONE NUMBER** *(include area code)* (937) 255-3636, x4281; gilbert.peterson@afit.edu