Steering Symbolic Execution to Less Traveled Paths

You Li* Zhendong Su+ Linzhang Wang* Xuandong Li*

*State Key Laboratory for Novel Software Technology
Department of Computer Science and Technology
Nanjing University, Nanjing, 210023, China

+University of California, Davis, USA

leo86@seg.nju.edu.cn su@cs.ucdavis.edu lzwang@nju.edu.cn lxd@nju.edu.cn

Abstract

Symbolic execution is a promising testing and analysis methodology. It systematically explores a program's execution space and can generate test cases with high coverage. One significant practical challenge for symbolic execution is how to effectively explore the enormous number of program paths in real-world programs. Various heuristics have been proposed for guiding symbolic execution, but they are generally inefficient and ad-hoc. In this paper, we introduce a novel, unified strategy to guide symbolic execution to less explored parts of a program. Our key idea is to exploit a specific type of path spectra, namely the *length-n subpath program spectra*, to systematically approximate full path information for guiding path exploration. In particular, we use frequency distributions of explored length-n subpaths to prioritize "less traveled" parts of the program to improve test coverage and error detection. We have implemented our general strategy in KLEE, a state-of-the-art symbolic execution engine. Evaluation results on the GNU Coreutils programs show that (1) varying the length n captures program-specific information and exhibits different degrees of effectiveness, and (2) our general approach outperforms traditional strategies in both coverage and error detection.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—*Reliability*, *Validation*; D.2.5 [*Software Engineering*]: Testing and Debugging—*Symbolic execution*

General Terms Algorithms, Experimentation, Verification

Keywords less traveled, path spectra, symbolic execution

OOPSLA '13, October 29–31, 2013, Indianapolis, Indiana, USA.

http://dx.doi.org/10.1145/2509136.2509553

1. Introduction

Testing is one of the most important software validation techniques, and arguably the most practical and widely-used. However, traditional approaches, such as random testing [14, 19] and manual testing, although useful, can be ineffective [15, 25, 31]. Symbolic execution [10, 22] is another classical technique for software testing and analysis. It can be used for systematically testing a program and test input generation with high coverage. Instead of using concrete input, symbolic execution uses symbolic values as input and explores a program's execution space. When symbolic execution encounters a branch condition, it forks the execution state, following both branch directions and updating the corresponding path constraints on the symbolic input. When it reaches a program exit or hits an error, the current path constraint will be solved to find a concrete test case that drives program execution to this program location. However, in practice, a program may have a large or even infinite number of paths because of conditionals, loops, and recursions. Thus, one key issue for making symbolic execution practical is *how to guide it toward more "profitable" paths*. To this end, various strategies have been proposed to guide and optimize symbolic exploration [5, 7, 17, 24, 32]. However, these path guidance techniques for symbolic execution are generally ineffective and ad-hoc.

In this paper, we introduce a *novel, unified methodology to guide symbolic execution* to explore programs "more effectively." At the high-level, we aim to explore "less traveled paths" of a program as it can help improve test coverage and locate more defects. Although most commonly adopted in practice, it is folklore that statement and branch coverage metrics may not accurately reflect how thorough a program has been exercised by a set of test cases. The recent work on Csmith [33] gives more concrete evidence. Csmith is a random C program generation tool specifically developed to find bugs in C compilers. The randomly generated C programs yield roughly the same statement and branch coverage as the test suites in the tested compilers (*e.g.*, GCC and Clang/LLVM), but they triggered a few hundred new bugs in GCC

and Clang/LLVM alone. Thus to measure "less traveled", a more flexible and general notion is needed.

Our *key conceptual idea* is to adapt a variant of path coverage, the *length-n subpath coverage* (where each subpath has n branches). Program profiling [4, 13, 27] has been effectively used to understand the behavior of a program. It can also be applied to measure test coverage using different program spectra. We propose the use of length-n subpath spectra to uniformly measure coverage and guide symbolic path exploration. Indeed, we use statistical analysis of already covered length-n subpaths to systematically steer symbolic execution to not yet or less explored parts of the program. The use of length-n subpaths offers a natural spectrum of program coverage. One extreme is the nontrivial degenerative case of branch coverage, when $n = 1$. The other extreme maps to complete path coverage, when $n = \infty$ (*i.e.* unbounded). We propose to use length-n subpath spectra to help prioritize program paths for exploration. Different choices of n give a systematic coverage, and we show that they exhibit different, yet complementary benefits.

In more detail, we maintain a priority queue P to store subpath spectra information on the explored parts of the program. For an explored (sub)-path $\pi = \langle s_0, s_1, \ldots, s_k \rangle$, each $s_i (0 \leq i \leq k)$ corresponds to a branch location and the direction taken in the execution. We compute π's length-n subpath fragments (*i.e.*, "n-grams") and store each in P along with its frequency information. When we need to decide which pending paths to explore, we examine P and select a pending path π' with the lowest frequency length-n subpath at its frontier. We terminate the path reaching the exit point or hitting an error. This process repeats until either the program has been fully explored, or we have reached a preset time limit or coverage threshold.

We have implemented our general strategy in KLEE [7], a state-of-the-art symbolic execution tool. To evaluate the effectiveness of our strategy, we ran KLEE on the GNU Coreutils programs [11] by varying the guidance strategies, including our own and the most common traditional strategies built into KLEE. We compared the quality of the generated test cases by measuring their statement coverage and defect detection capabilities. The results show that our strategy can guide symbolic execution to cover the program faster and find more bugs than the evaluated traditional strategies. In summary, we have the following key findings: 1) different choices of n exhibit different behavior and capture program-specific information; and 2) the proposed unified strategy significantly outperforms the other strategies.

In this paper, we make the following main contributions:

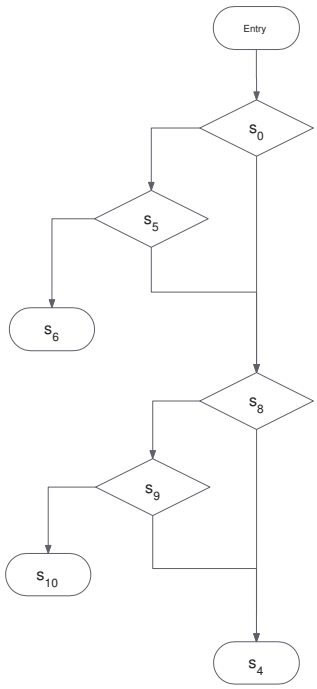
- We introduce the novel concept of using subpath program spectra as a unified strategy to guide symbolic execution and present the details on how to realize the strategy.
- We implement the general strategy within KLEE and extensively evaluate its effectiveness against common traditional strategies.

```
      main (x, y) {
s0:     if (x > y)
s1:       x = f(x);
        else
s2:       ;
s3:     g (x, y);
s4:     return;
      }

      int f (a) {
s5:     if (a > 0)
s6:       ABORT;
        else
s7:       return -a;
      }

      g (a, b) {
s8:     if (a == 0)
s9:       if (b == 0)
s10:        ABORT;
          else
s11:        ;
        else
s12:      print a/b;
s13:    return;
      }
```

(a) Example Program



(b) Control Flow

Figure 1. An example program.

The rest of the paper is structured as follows. Section 2 provides details on how to use length-n subpath program spectra to guide symbolic execution. We also give necessary background on symbolic execution and use an example to illustrate the benefits of our approach. In Section 3, we describe the implementation and evaluation of our approach. We survey related work in Section 4 and conclude in Section 5.

2. Subpath-Guided Path Exploration

This section details our proposed path exploration strategy. We start with some necessary background on symbolic execution and path spectra.

2.1 Symbolic Execution Background

The concept of symbolic execution [22] was proposed by King in 1976. It is a practical testing and analysis approach that utilizes and combines testing and program verification. Recent years have witnessed an impressive amount of work [1, 5, 7, 8, 17, 18, 24, 26, 32] on building practical symbolic execution techniques and tools for improving test input generation and bug finding. The key idea of symbolic execution is to substitute the actual data with symbolic value as the input data. The corresponding program operations are replaced with symbolic expressions, so the output of the program can be represented as formulas on the symbolic input.

To illustrate the procedure of symbolic execution, we use the example program in Figure 1. At the entry point of the program, symbolic execution generates an original execution

state ES. An execution state maintains 1) the symbolic values (x and y), 2) the corresponding symbolic expressions (such as $x = f(x)$ in s_1), and 3) symbolic path constraint PC of the input (such as $x > y$ in s_0), which can represent a specific path of the program. The symbolic execution engine then executes the program with ES. When it encounters a conditional statement "**if** (e) s_1 **else** s_2" (such as s_0 in the program), it generates a new execution state ES'. The new state ES' copies all the information from ES, and the path constraints are updated accordingly: the path constraint of ES' is updated to $PC \wedge \neg(e)$, while the path constraint of ES is updated to $PC \wedge (e)$. Therefore, ES and ES' can explore both sides of the conditional statement. As more and more execution states are generated, different strategies are applied to decide which execution state is desirable to be executed next at each fork (*i.e.*, a conditional branch either from conditionals or loops in the source program).

During this process, a constraint solver is used to check the satisfiability of the path constraint in an execution state. If it is unsatisfiable, the corresponding path is discarded from further exploration as it is infeasible. When an execution state reaches an exit point of the program or hits an error, the path exploration terminates and the corresponding path constraint is solved to find a concrete input, if any. Assuming the latest execution state is always selected, under the Depth-First Search (DFS) strategy, the path $s_0 s_2 s_3 s_{12} s_{13} s_4$ is the first case generated and the path constraint is $\neg(x > y) \wedge \neg(x == 0)$, then the constraint solver can generate a concrete input $(1, 0)$ (*i.e.*, $x = 1$ and $y = 0$). This input is typically equivalent to many other test cases following the same paths. When we execute the program with this concrete input, it can repeat the same path as the execution state did originally via symbolic execution.

However, in realistic programs, loops or recursion can lead to a large or even infinite number of paths (in theory). In most situations, one has to set a time limit on symbolic execution. Thus, it is important to guide symbolic execution to select profitable states to explore during its search. Consider KLEE, a state-of-the-art symbolic execution engine. It has several built-in traditional search strategies [7]:

- *Depth-First Search (DFS)* always selects the latest execution state among all the states to explore next. This strategy has little overhead in selecting a state, however, it typically explores fewer parts of the program and gets stuck when it encounters a *tight loop* — a loop with few statements that iterates many times.
- *Random State Search (RSS)* randomly selects a pending state to explore. This strategy can explore the program more uniformly and avoid the situation with tight loops with a symbolic condition creating new states rapidly (fork bombing). The problem of RSS is that it may repeatedly generate test cases that have the same effect as those generated earlier.
- *Random Path Selection (RPS)* uses a binary execution tree to record the information on explored parts of the program, where the leaves are current states and the internal nodes are the forks. It selects states by traversing the execution tree from the root and randomly picks a direction when it encounters a branch until it reaches a leaf. With this strategy, those states high up in the execution tree have greater chance to be chosen because these states have fewer constraints to satisfy and may be more likely to explore uncovered parts of the program. RPS can avoid the fork bombing problem affecting RSS, but as RSS, it may also repeatedly generate similar test cases.
- *Coverage-Optimized Search (COS)* uses heuristics to compute which state has better chance to cover new code very soon. It calculates a weight for the states to be chosen and selects a state *w.r.t.* the weight. Various factors, such as the minimum distance to an uncovered instruction and the query cost, are taken into account to calculate the weight for each state. This strategy is not general and may not perform well for every program.

Using search heuristics, such as the ones above, is one possible approach to deal with the problem of path explosion in symbolic execution. Traditional methods mostly perform blind, random search, thus cannot explore the program uniformly and thoroughly.

2.2 The Length-n Subpath Program Spectra

To effectively guide symbolic execution, we believe that the "less traveled" parts of the program are more profitable. Exploring these parts can generate test cases with higher coverage and better defection detection capabilities. To capture which executions may lead to "less traveled" parts, we need to have means to approximate the behavior of the program. To this end, this paper proposes a novel application of program spectra to help approximate program behavior and guide path exploration.

Program profiling has been used to understand dynamic program behavior. It counts occurrences of different runtime events during a program's execution [4]. These events can be paths, basic blocks, control-flow edges, *etc.* Profiling based on different types of events provides various program spectra that can help characterize the program's execution and provide signatures of a program's runtime behavior [27]. Below we list a few of the most widely used program spectra:

- *Branch Hit Spectra*: Recording the executed conditional branches.
- *Branch Count Spectra*: Recording the execution count for each conditional branch.
- *Complete Path Spectra*: Recording the complete paths that were explored.
- *Path Spectra*: Recording the explored intraprocedural loop-free paths.

- *Path Count Spectra*: Recording the executed count for each intraprocedural loop-free path.

The above profile information can be used in many ways, such as compiler optimizations [2, 3], regression testing [20], coverage measuring, and many other applications that need to analyze program behavior. Reps *et al.* [27] use path spectra to deal with the "Year 2K Problem." They compared spectra from different runs of the same program to locate date-dependent computations. An empirical study [20] shows various types of spectra differences highly correlate with the exposure of regression faults. However, there does not exist a single program spectrum that can discover all program behavior differences. Thus, using different program spectra can help us understand a program's behavior more comprehensively from different aspects.

From the above discussion, different program spectra provide different perspectives to understand a program's execution behavior. We can use program spectra to measure the effect of different test generation techniques. Complete path coverage is a highly precise metric to measure the quality of a test suite, however, it is impractical to obtain high coverage as we need a large number of test cases to cover possible complete paths, many times even infinite. On the other hand, branch coverage, statement coverage or function coverage are too shallow to accurately capture the effectiveness of test suites. The random C program generation tool Csmith [33] has been used to discover hundreds of new bugs in mainstream C compilers, such as GCC and Clang/LLVM. However, as the authors have noted, the generated test cases and the compilers' test suites are very similar with respect to the three coverage metrics mentioned above.

To fill the gap between complete path coverage and branch coverage, we adopt the concept of *length-n subpath* program spectra. It is a variant of complete path coverage. A complete path is a unique sequence of branch conditions from the entry of a program to its exit, and we can represent it as $\pi = \langle s_0, s_1, \ldots, s_k \rangle$ where each s_i corresponds to a conditional branch (*i.e.*, fork) and the branch direction taken in the program. We represent a length-n subpath as $\langle s_{i+1}, s_{i+2}, \ldots, s_{i+n} \rangle$, which is a consecutive sub-sequence from a complete path.

By varying n, we obtain a spectrum of modeling precision. When $n = 1$, it degenerates to branch coverage. At the other extreme, we obtain complete path coverage when $n = \infty$ (*i.e.*, unbounded).

2.3 Subpath-Guided Search

To measure which parts are "less traveled", we propose to use length-n subpath. However, it is time-consuming to build the whole length-n subpath matrix during symbolic execution. Instead, we use statistical analysis of the already covered length-n subpaths to decide which execution state(s) may lead the path exploration to the "less traveled" parts. We call this general, uniform strategy *Subpath-Guided Search* (**SGS**).

Algorithm 1: Subpath-Guided Search Strategy (*Part 1*)

```
   Initialization:
 1: PriorityQueue<pathSegmentCount> P;
 2: Vector<executionState> ESVector;
 3: executionState initialState;
 4: Integer pathSegmentLength;
   Begin symbolicExecution:
 5: ESVector.add(initialState);
 6: while ESVector.size>0 || !TIME_OUT do
 7:   executionState ES = selectState();
 8:   P[ES.π]++;
 9:   while ES.instructionType != FORK || EXIT &&
      !FoundError do
10:     ES.executeInstruction();
11:   end while
12:   if ES.instructionType = EXIT || FoundError then
13:     generateTestCase();
14:     ESVector.remove(ES);
15:     continue;
16:   end if
17:   if ES.instructionType = FORK then
18:     instruction Fr = ES.currentInstruction;
19:     ES2 = new executionState(ES);
20:     ES.newNode = Fr(T)
21:     updatePathSegment(ES);
22:     if !P.contains(ES.π) then
23:       P.add(ES.π,0);
24:     end if
25:     ES2.newNode = Fr(F);
26:     updatePathSegment(ES2);
27:     if !P.contains(ES2.π) then
28:       P.add(ES2.π,0);
29:     end if
30:     ESVector.add(ES2);
31:   end if
32: end while
   End symbolicExecution
```

The main algorithm is shown in Algorithm 1 & Algorithm 2. We use a structure $e = \langle \pi, f \rangle$ for statistical analysis, where π is the subpath (of length n), f is the frequency of π in the explored parts of the program, which is calculated by counting the number of times that π has been explored before. A priority queue P is maintained and the subpath with the lowest frequency is at the top of the queue. For each execution state ES, a corresponding subpath π is maintained, which is composed of the latest n branch conditions executed of ES. The execution state with the lowest count of π is selected to continue the symbolic execution (Algorithm 1, Line 7). We break ties, if any, at random. When an execution state ES_k (ES in Algorithm 1) is picked, the frequency of the corresponding subpath π_k (ES.π in Algorithm 1, Line 8) is increased by 1. As described in Section 2.1, at every fork point

Steps	Pending Path	Priority Queue	Path to Pick	Generate Case
1	1:s_{0t} 2:s_{0f}	(*s_{0t},0),(*s_{0f},0)	2	
2	1:s_{0t} 2:$s_{0f}s_{8t}$ 3:$s_{0f}s_{8f}$	(*s_{0t},0),(*s_{8t},0),(*s_{8f},0),(s_{0f},1)	3	$s_{0f}s_{8f}s_4$
3	1:s_{0t} 2:$s_{0f}s_{8t}$	(*s_{0t},0),(*s_{8t},0),(s_{8f},1),(s_{0f},1)	2	
4	1:s_{0t} 2:$s_{0f}s_{8t}s_{9t}$ 3:$s_{0f}s_{8t}s_{9f}$	(*s_{0t},0),(*s_{9t},0),(*s_{9f},0),(s_{8t},1), (s_{8f},1),(s_{0f},1)	3	$s_{0f}s_{8t}s_{9f}s_4$
5	1:s_{0t} 2:$s_{0f}s_{8t}s_{9t}$	(*s_{0t},0),(*s_{9t},0),(s_{9f},1),(s_{8t},1), (s_{8f},1),(s_{0f},1)	2	$s_{0f}s_{8t}s_{9t}s_{10}$
6	1:s_{0t}	(*s_{0t},0),(s_{9t},1),(s_{9f},1),(s_{8t},1), (s_{8f},1),(s_{0f},1)	1	
7	1:$s_{0t}s_{5t}$ 2:$s_{0t}s_{5f}$	(*s_{5t},0),(*s_{5f},0),(s_{0t},1),(s_{9t},1), (s_{9f},1),(s_{8t},1),(s_{8f},1),(s_{0f},1)	2	
8	1:$s_{0t}s_{5t}$ 2:$s_{0t}s_{5f}s_{8t}$ 3:$s_{0t}s_{5f}s_{8f}$	(*s_{5t},0),(s_{5t},1),(s_{0t},1),(s_{9t},1), (s_{9f},1),(*s_{8t},1),(*s_{8f},1),(s_{0f},1)	3	$s_{0t}s_{5t}s_6$
9	1:$s_{0t}s_{5f}s_{8t}$ 2:$s_{0t}s_{5f}s_{8f}$	(s_{5t},1),(s_{5t},1),(s_{0t},1),(s_{9t},1), (s_{9f},1),(*s_{8t},1),(*s_{8f},1),(s_{0f},1)	2	$s_{0t}s_{5f}s_{8f}s_4$
...	...	...	...	...

Table 1. Length-1 subpath-guided search.

Algorithm 2: Subpath-Guided Search Strategy (*Part 2*)

```
Begin selectState
 1: Vector<executionState> selectSet;
 2: for 0<i<ESVector.size() do
 3:    if P(ESVector[i].pathSegment)=P.lowest then
 4:       selectSet.add(ESVector[i]);
 5:    end if
 6:    i++;
 7: end for
 8: Integer random = randomInteger() mod selectSet.size();

 9: return selectSet[random];
End selectState
10:
Begin updatePathSegment(executionState ES)
11: ES.π.add(ES.newNode);
12: if ES.π.length > pathSegmentLength then
13:    ES.π.removeFirstBranchCondition();
14: end if
End updatePathSegment(executionState ES)
```

Fr, a new execution state ES'_k is generated (Algorithm 1, Line 19). The states ES_k and ES'_k (ES2 in Algorithm 1, Line 19) will explore both sides of the fork, so we update the corresponding subpaths of ES_k and ES'_k to $\pi_k + Fr(T)$ and $\pi_k + Fr(F)$ (Algorithm 1, Lines 20 and 25). $Fr(x)$ denotes a fork point Fr with the direction it chooses, where x can be either T or F. If the length of the new subpath is larger than n, the first branch condition will be removed (Algoritm 2, *updatePathSegment*). Then we check the priority queue P. The new subpath with a 0 count will be added to P if it has not been encountered earlier (Algorithm 1, Lines 23 and 28). After updating the priority queue, we pick a new execution state with the lowest count of π (Algoritm 2, *selectState*). This process repeats until all the execution states have terminated or the symbolic exploration times out. To illustrate the benefits of **SGS**, we use length-1 subpath to guide the symbolic exploration to generate test cases for the example program in Figure 1 and compare it with the depth-first search strategy. To have a fair comparison, we assume that our strategy breaks ties by selecting the latest execution state as DFS does. There are 7 complete paths in the example program. To generate test cases to cover all the statements, DFS needs to explore all 7 paths. The procedure of length-1 subpath guided search is shown in Table 1. Node s_{it} represents a branch node s_i with the true direction taken, while s_{if} denotes a branch node s_i taking the false direction. The column "Pending Path" records the corresponding paths of the unterminated execu-

Steps	Pending Path	Priority Queue	Path to Pick	Generate Case
1	1:s_{0t} 2:s_{0f}	(*s_{0t},0),(*s_{0f},0)	2	
2	1:s_{0t} 2:$s_{0f}s_{8t}$ 3:$s_{0f}s_{8f}$	(*s_{0t},0),(*$s_{0f}s_{8t}$,0),(*$s_{0f}s_{8f}$,0),(s_{0f},1)	3	$s_{0f}s_{8f}s_4$
3	1:s_{0t} 2:$s_{0f}s_{8t}$	(*s_{0t},0),(*$s_{0f}s_{8t}$,0),($s_{0f}s_{8f}$,1),(s_{0f},1)	2	
4	1:s_{0t} 2:$s_{0f}s_{8t}s_{9t}$ 3:$s_{0f}s_{8t}s_{9f}$	(*s_{0t},0),(*$s_{8t}s_{9t}$,0),(*$s_{8t}s_{9f}$,0),($s_{0f}s_{8t}$,1), ($s_{0f}s_{8f}$,1),(s_{0f},1)	3	$s_{0f}s_{8t}s_{9f}s_4$
5	1:s_{0t} 2:$s_{0f}s_{8t}s_{9t}$	(*s_{0t},0),(*$s_{8t}s_{9t}$,0),($s_{8t}s_{9f}$,1),($s_{0f}s_{8t}$,1), ($s_{0f}s_{8f}$,1),(s_{0f},1)	2	$s_{0f}s_{8t}s_{9t}s_{10}$
6	1:s_{0t}	(*s_{0t},0),($s_{8t}s_{9t}$,1),($s_{8t}s_{9f}$,1),($s_{0f}s_{8t}$,1), ($s_{0f}s_{8f}$,1),(s_{0f},1)	1	
7	1:$s_{0t}s_{5t}$ 2:$s_{0t}s_{5f}$	(*$s_{0t}s_{5t}$,0),(*$s_{0t}s_{5f}$,0),($s_{8t}s_{9t}$,1),($s_{8t}s_{9f}$,1), ($s_{0f}s_{8t}$,1),($s_{0f}s_{8f}$,1),(s_{0t},1),(s_{0f},1)	2	
8	1:$s_{0t}s_{5t}$ 2:$s_{0t}s_{5f}s_{8t}$ 3:$s_{0t}s_{5f}s_{8f}$	(*$s_{0t}s_{5t}$,0),(*$s_{5f}s_{8t}$,0),(*$s_{5f}s_{8f}$,0),($s_{8t}s_{9t}$,1), ($s_{8t}s_{9f}$,1),($s_{0f}s_{8t}$,1),($s_{0f}s_{8f}$,1),(s_{0t},1), (s_{0f},1)	3	$s_{0t}s_{5f}s_{8f}s_4$
9	1:$s_{0t}s_{5t}$ 2:$s_{0t}s_{5f}s_{8t}$	(*$s_{0t}s_{5t}$,0),(*$s_{5f}s_{8t}$,0),($s_{5f}s_{8f}$,1),($s_{8t}s_{9t}$,1), ($s_{8t}s_{9f}$,1),($s_{0f}s_{8t}$,1),($s_{0f}s_{8f}$,1),(s_{0t},1), (s_{0f},1)	2	
10	1:$s_{0t}s_{5t}$ 2:$s_{0t}s_{5f}s_{8t}s_{9t}$ 3:$s_{0t}s_{5f}s_{8t}s_{9f}$	(*$s_{0t}s_{5t}$,0),($s_{5f}s_{8t}$,1),($s_{5f}s_{8f}$,1),(*$s_{8t}s_{9t}$,1), (*$s_{8t}s_{9f}$,1),($s_{0f}s_{8t}$,1),($s_{0f}s_{8f}$,1),(s_{0t},1), (s_{0f},1)	3	$s_{0t}s_{5t}s_6$
...	...	...	...	...

Table 2. Length-2 subpath-guided search.

tion states, the column "Priority Queue" shows the count and position changes of the length-1 subpath, the structure $e = \langle \pi, f \rangle$ with a $*$ in the priority queue indicates that there are unterminated execution states which have the corresponding subpath π. When an execution state reaches the exit of the program, the corresponding test case will be generated and shown in the rightmost column, then the execution state is terminated and removed from the pending path.

In the first 7 steps, subpath-guided search has the same selection as DFS. Three test cases are generated, which cover all the sub-paths started from s_8. Three unterminated execution states with the path $s_{0t}s_{5t}$, $s_{0t}s_{5f}s_{8t}$ and $s_{0t}s_{5f}s_{8f}$ can be chosen. In step 8, the branch condition s_{5t} becomes the "least traveled" part compared to s_{8f} and s_{8t}, subpath-guided search then drives symbolic execution to explore that branch and the case with the path $s_0s_5s_6$ is generated as the 4th case, while DFS will select the third execution state. In step 9, for all the count of the length-1 subpath is 1, subpath-guided search selects the latest execution and generates the test case for $s_{0t}s_{5t}s_{8f}s_4$. It takes only 5 cases to cover all the statements in the example program with the guidance of length-1 SGS.

Different lengths can provide different effects for path guidance. Table 2 shows the procedure of the length-2 SGS. Same as the length-1 SGS, the length-2 SGS generates 5 cases to cover all the statements, and the same selection in the first 7 steps. The length-2 SGS counts the frequencies of the length-2 sub-paths. In step 8, the execution state corresponding to the path $s_{0t}s_{5f}s_{8f}$ is selected because the sub-path $s_{5f}s_{8f}$ has never been explored before. On the other hand, in length-1 SGS, the sub-path s_{8f} is explored, and the execution state corresponding to the path $s_{0t}s_{5t}$ is selected. The length-2 SGS considers more contextual information, which guides symbolic execution to possibly explore moree directions in a program. We see that the length-2 SGS takes one extra step than length-1 SGS to generate cases covering all statements of the example program.

The subpath-guided search can also avoid the fork bombing problem, if we replace s_{11} with "s'_{11}: **while** (a < 10) a++;", the DFS strategy will keep creating new execution states in s'_{11} and lead to the starvation of the other states. Conversely, in our strategy, when both conditions in s'_{11} are executed once, it prefers those states with a zero count of the corresponding subpath to the loop entrance. When all the

zero count subpaths are explored, the loop entrance node will be explored again.

2.4 Discussions

Subpaths with different lengths provide various program spectra and conceptually capture different levels of "less traveled" program parts. In our approach, shorter subpaths are more similar to using a single branch condition, and thus may guide symbolic execution to cover those uncovered statements with higher probabilities. However, shorter subpaths contain less contextual information of the explored parts, thus some execution states may share the same corresponding subpaths. In this case, symbolic exploration may ignore some special execution states that lead to particular program parts. On the other hand, longer subpaths may divide the execution states with smaller granularity, but some redundant test cases may be generated. In the next section, we empirically explore such trade-offs and show how to synergistically combine the benefits of different choices of n.

3. Evaluation and Analysis

This section presents our evaluation and analysis of the proposed strategy. It describes details of our evaluation design, setup, and results. In particular, we show that, on realistic programs, our strategy systematically captures program-specific information and exhibits varying levels of effectiveness with different choices of n, and it performs significantly better than traditional strategies in terms of test coverage and error detection.

3.1 Evaluation Design and Setup

We have implemented the proposed length-n subpath guided search in KLEE [7, 23], a state-of-the-art symbolic execution engine built on top of the LLVM compiler infrastructure. KLEE can process a large number of concurrent states and has strong support for handling interactions with the external environment [9]. KLEE also provide different built-in search strategies, including traditional DFS, random state search, and some other heuristic search strategies. Adding our unified search strategy to KLEE, we are able to directly compare their effectiveness and trade-offs in test case generation not on toy programs, but realistic programs from GNU Coreutils.

Research Questions. Through empirical analysis, we hope to answer the following key research questions:

(R1) What impact do different choices of n have? Can they be effectively combined?

(R2) How does our strategy compare with the traditional strategies?

In particular, for **(R1)**, we aim to understand what different characteristics the test suites generated from different n's have in terms of coverage and error detection. We will also understand whether there is a uniformly best n. As for **(R2)**, we seek to understand how our strategy compares to traditional strategies also in terms of coverage and error detection. We have designed and ran a set of experiments to answer these questions.

Program	ELOC	Mutant	Program	ELOC	Mutant
base64	3989	1204	nohup	3875	
basename	4026	356	od	4463	6762
cat	3953		paste	3837	1525
chcon	4343	1313	pathchk	3857	1078
chgrp	4278	672	pr	4626	
cksum	3983	1066	printenv	3881	
comm	3997	820	printf	4251	2745
csplit	8589		pwd	3969	
cut	4195	2821	readlink	4154	284
date	5688		rm	4560	
dd	4734	5450	rmdir	3892	454
df	4314		seq	3927	
dircolors	4093	1527	setuidgid	3878	548
dirname	3889	209	shuf	4508	
du	5790	2168	sleep	4199	381
echo	3884		split	4428	2169
env	3937	334	stat	4210	
expand	3916	1144	stty	4718	
expr	9565	2333	sum	4068	954
factor	3896		sync	3919	89
false	3897		tail	4495	
fmt	3860		tee	3966	593
fold	3891	1064	test	3577	
groups	4002	232	touch	4744	1660
head	4170		tr	4150	6640
id	4067		true	3888	
join	4617		tsort	3856	1120
kill	3919		tty	3847	
link	3829	272	uname	3810	
ln	4200		unexpand	3903	1336
logname	3902	102	uniq	4048	
ls	6549		unlink	3865	186
mkdir	4213	408	uptime	3896	
mkfifo	3959	343	users	3907	
mknod	3840	868	wc	4075	2205
mktemp	4101		whoami	3856	
nice	4010	764	yes	3901	
nl	10037	1591			

Table 3. Test subjects from GNU Coreutils.

Evaluation Subjects. To measure the effectiveness of different length-n subpath guided search strategy. Following KLEE, we have selected GNU COREUTILS utilities as the test subjects. They are the basic file, shell and text manipulation utilities on the GNU operating system [11]. The version of COREUTILS that we use is 6.11 (as from the tutorials from KLEE's website). All the experiments were ran on a server with Intel(R) Xeon(R) X7542 CPU (18 cores, 2.67GHz). The operating system is Ubuntu 10.04. The programs we use are shown in Table 3. The column "ELOC" shows the size of the programs in terms of the number of executable lines of

code (ELOC). ELOC shows the total executable lines of the final executable we ran KLEE on after optimization. Because KLEE may invoke library code to execute some parts of the program we test, we also include the library code when measuring the raw size of the programs, again following KLEE [7]. From the table, we can see that the size for most programs ranges between 3K to 4K, while six programs have more then 5K lines. This statistical information shows that the programs are not tiny, toy problems. In addition, we have applied mutant testing [12] to evaluate the effectiveness of the strategies. The column "mutant" lists those programs selected for mutant testing and the corresponding number of mutants for each selected program.

Traditional Strategies. KLEE provides four main search heuristics: Depth-First Search (DFS), Random State Search (RSS), Random Path Selection (RPS), and Non Uniform Random Search (NURS). NURS randomly selects states according to a given distribution based on some properties. In our evaluation, we choose the following properties: whether it covers new instructions (covnew), the depth (depth), the Instr-Count heuristic (icnt), and the minimum distance to an uncovered instruction (md2u). The NURS is interleaved with Random Path Selection (RPS), while the default heuristics used by KLEE are random-path interleaved with nurs:covnew. For subpath-guided search, we choose the length to range over $\{1, 2, 4, 8\}$ for comparison.

We ran KLEE on each program with the command:

```
./run<program>  --search-strategy
                --max-time 3600
                --sym-args 0 3 10
                --sym-files 2 8
```

The option `--max-time 3600` sets the time limit to $3,600$ seconds. The option `--sym-args 0 3 10` allows KLEE to replace 0 to 3 command line arguments of the program with at most length 10. The option `--sym-files 2 8` tells KLEE to make at most 2 standard input symbolic files with maximal length 8. It is noted that KLEE can generate test cases for those unterminated execution states after the time limit. However, for some strategies, there are many unterminated states, which may consume much of the algorithms' time and lead to inaccurate time limits. To be fair in our comparison, we ignore the test cases corresponding to unterminated execution states and compare different strategies with the test cases generated within the given time limit.

3.2 Test Coverage Results

Our first set of evaluations focuses on evaluating the test coverage (in particular statement coverage) of the generated test cases under different search strategies. We use the tool **gcov** to compute the statement coverage information. It can be used in conjunction with GCC and generate executables to profile an instrumented program. KLEE provides a tool to replay the test cases on the corresponding executable and **gcov** can calculate statement coverage. When measuring coverage, we only consider the code in the program itself and do not include the library code since it is invoked from many programs to avoid counting them multiple times. By default, KLEE generates one test case for each terminated path. However, for larger programs, it is quite costly to compute and re-execute the test cases covering explored parts of the program. Since our goal is to measure statement coverage, we use the `--only-output-states-covering-new` option on the KLEE command line, so that KLEE only outputs test cases for the paths covering new instructions in the main utility code (or hit an error).

We have selected 75 programs in COREUTILS for coverage comparison. Table 4 shows the result of the comparison. We show the distribution of the coverage under each search strategy. In addition, the row "Avg." shows the average coverage information across all programs. The row "Best" lists the total number of programs with *best coverage* (for ties, we count 1 for each).

Result 1: SGS yields higher coverage. From Table 4, we observe that subpath-guided search (SGS) performs better than the other strategies, across all choices of n. The NURS strategies do not perform well. One possible explanation is that we ignore the cases corresponding to unterminated states. The results demonstrate the disadvantages of the random choices: sometimes they cannot drive the execution states to reach the exit to effectively generate test cases. The random path search (RPS) strategy has similar results as length-8 (*i.e.*, $n = 8$) subpath guided search, which shows that for the COREUTILS programs, length-8 subpaths seem long enough to approximate complete paths.

Result 2: No uniform best n for SGS. We observe that although the length-2 subpath-guided search shows the best results compared to the other strategies (both the highest average coverage and the most number of highest covered programs), it is still not the best uniform n for about half of the programs. Different subpath lengths exhibit different characteristics, so we need a method to unify the guidance of different lengths.

As we discussed in Section 2, different length subpaths profile different spectra of a program. They can provide different understanding of a program's behavior and guide symbolic execution to different parts of the program. Combining the results of SGS with different subpath lengths may provide a more comprehensive exploration of the program. To find out the power of combining different lengths' results, we ran KLEE on the benchmarks with the guidance of each length subpath for 15 minutes. We replayed the test suites individually and then combined all the test cases to see their effects.

Result 3: Combined SGS performs the best. Table 5 presents the coverage distribution of the individual subpath-guided search and the combined strategy. We observe that individual strategies can quickly achieve high statement

Cov.	SGS $n=1$	SGS $n=2$	SGS $n=4$	SGS $n=8$	RSS	DFS	RPS	NURS covnew	NURS depth	NURS icnt	NURS md2u
90-100%	18	18	21	13	14	11	17	11	16	11	11
80-90%	4	12	9	10	5	3	9	3	4	5	4
70-80%	14	16	12	13	5	6	12	2	12	3	3
60-70%	17	11	12	15	12	11	12	4	14	5	6
60%-	22	18	21	24	39	44	25	55	29	51	51
Avg.(%)	69.67	72.87	72.35	66.95	60.56	56.12	68.38	46.87	64.35	51.72	50.50
Best	25	38	29	24	20	16	23	14	21	14	15

Table 4. The coverage distribution of the COREUTILS programs under the guidance of different search strategies.

Cov.	$n=1$	$n=2$	$n=4$	$n=8$	**Com**
90-100%	9	15	11	8	**25**
80-90%	10	13	12	12	10
70-80%	19	10	12	17	19
60-70%	9	11	8	10	9
60%-	28	26	31	28	12
A.Cov(%)	66.06	68.88	66.22	65.88	**80.67**
Best	24	27	25	23	**57**

Table 5. The coverage distribution of subpath guided-search with 15 minutes time limit and the result of the combined cases. The row "Best" is the comparison result of the test cases generated with individual strategies under 60 minutes time limit, and the combined test cases of those in 15 minutes.

coverage and saturate, and the combined strategy in 15 minutes can already outperform all the other strategies in KLEE, each with 60 minute time limit. The comparison results of the total number of best coverage among the individual strategies in 1 hour and the combined strategy in 15 minutes each are shown in the row "Best". We can see that the combined strategy has the best coverage in almost 80% of the programs and the average coverage is also significantly higher.

We also examined the speed to find test cases covering new statements or branches by tracing the number of terminated states when each test case is generated and replaying the cases one by one to see the trend of increasing coverage. In some bigger programs, length-1 SGS terminates fewer execution states when generating new test cases than the longer lengths. However, when the coverage reaches a certain level, it becomes difficult to find new test cases, while the longer n's can still guide path exploration to find new test cases and ultimately gain higher coverage.

Result 4: SGS yields more bug reports. When KLEE explored the programs, it also issued some bug reports. Table 6 summarizes this information from the first set of evaluations in Table 4. There are four kinds of bugs reported [23]:

- *Model*: KLEE does not support certain program states. For example, KLEE cannot support symbolic sizes to malloc.
- *Exec*: Some problems prevented KLEE from executing the program, such as unknown instructions, a call to an invalid function pointer, or inlined assembly.
- *Ptr*: Stores or loads of invalid memory locations.
- *External*: KLEE failed when invoking external functions with symbolic arguments.

In Table 6, we only show the programs where different search strategies yielded bug reports. The *Model*, *Exec*, and *External* bugs may be resulted from imprecise modeling in KLEE. However, to illustrate that our strategy was able to trigger some issues while the other strategies did not, we included all types of bugs in the table. We observe that subpath-guided search reported both more bugs and more types of bugs than the other strategies. Below we list some test cases that our strategies reported bugs, but the other strategies did not:

- **dir**: Both SGS $n=1$ and $n=2$ generated a *Ptr* error. The command line arguments "A // -c" (generated with $n=1$) and "-ccab A /" (generated with $n=2$) with a null directory A caused an out-of-bound pointer error in the library `readdir.c:33` from uclibc.
- **chcon**: SGS $n=1$ reported an *External* error. When we use the command line arguments "- –d –ra=g", KLEE failed on an external call to `fstatat()` in the Coreutils' library `fts.c:1394`.
- **shuf**: Both SGS $n=1$ and $n=2$ generated a *Model* error. KLEE reported that the command line arguments "-i 03-7@" (with $n=1$) and "-i3-8_" (with $n=2$) need to malloc with a symbolic argument, while KLEE does not support and has to concretize the argument.
- **test**: SGS $n=2$ reported an *Exec* execution error. KLEE cannot execute the program with the command line arguments " -t +01", which makes KLEE execute the function `syscall()` in the POSIX support code `fd.c:901` with symbolic arguments.

Programs \ Strategy	SGS $n=1$	SGS $n=2$	SGS $n=4$	SGS $n=8$	DFS	RSS	RPS	NURS covnew	NURS depth	NURS icnt	NURS md2u
chcon	*1,1										
chgrp	1,1	1,1	1,1	1,1		1,1	1,1	1,1	1,1	1,1	1,1
csplit	*1,2	1,1	1,1	1,1		1,1					
dir	*2,1	*2,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1		1,1
head	*1,2	1,1	*1,2	*1,2	1,1						
ls	1,1	*2,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1
od	1,3	1,4	*1,8	*1,8	1,1	1,6	1,4	1,2	1,3	1,2	1,2
printf	*1,1	*1,1			1,1						
rm	1,1	1,1	*1,2	1,1		*1,2	1,1	1,1	1,1	1,1	1,1
shuf	*2,1	*2,1									
split	1,1	*1,2	*1,2	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1
stat	*2,1	1,1	*2,1	*2,1	1,1	*2,1	1,1	1,1	1,1	1,1	1,1
test		*1,1									
unexpand	1,1	1,1	1,1	1,1		1,1	1,1		1,1	1,1	

Table 6. Bug reports from KLEE. The types of bugs are: Model, Exec, Ptr, and External. An empty cell indicates that the corresponding strategy did not report any errors. Each nonempty cell has two numbers: the first for the number of reported error types, and the second for the total number of error reports. A cell with a '*' indicates that the corresponding search strategy performed best in bug finding.

The error reports from different search strategies indicate that our strategy were able to trigger such issues but the other strategies did not. We can also observe that the guidance under different subpath lengths may lead to different bug reports.

Result 5: SGS has acceptable overhead. In this experiment, we observed that each search strategy has quite similar total number of explored paths as the other strategies on most programs. This indicates that the SGS strategy does not incur much time overhead. As for space overhead, KLEE ran out of memory and crashed for a few programs, mainly because the constraint solver consumed too much memory. This happened for almost all the search strategies and may be attributed to some specific paths of the programs. This means that our SGS strategy also has similar and acceptable space overhead compared to the other strategies.

3.3 Mutation Testing Results

To further compare the usefulness of the generated test cases under different search strategies, we applied mutation testing [12] to measure the quality of the generated test cases. In mutation testing, small changes are applied to a program **P** to generate a set of "faulty" programs called *mutants*. A test suite is then executed on the original program and the mutants. If the test data can detect the mutated code, we say the test suite kills a mutant. The ability to find bugs of a test suite can be measured by the number of mutants it kills.

We used a state-of-the-art mutation testing tool for C MILU [21] to generate mutants for the programs in COREUTILS. 21 pre-defined mutation operators are applied to make simple replacements to the programs in COREUTILS, such as replacing "++" with "–" or ">" with "<". Then we replayed the test cases generated under each search strategy on the original version and the mutants, compared their outputs to decide whether a mutant was killed by the test suite. Some programs in COREUTILS do not produce output or their output is environment related (such as `date` and `df`), it is difficult to tell whether the different outputs between original code and a mutant is caused by the test suite we used. In this case, we picked 40 of the programs for this mutant testing evaluation.

Table 3 shows the programs we chose, and for each program, its corresponding number of mutants. Because the size of each program is different, MILU generated different number of mutants for each program, ranging between 100 to around 6K. The total number of the mutants is 57,790. Unlike what we did in the statement coverage evaluation, test cases were generated for every terminated state. This is because the distribution of the small changes in the program may need some specific paths to show their effects, not only those test cases for paths that hit new instructions or branches that lead to different outputs in the original code and the mutants. Because the evaluation results from Section 3.2 show that the combined subpath-guided search strategy is the most effective in terms of statement coverage, we also added the combined strategy to our mutant testing evaluation.

The results are shown in Table 7. Since the number of mutants for the programs varies widely, we use three dimensions to present the results. The column "Total Kill" denotes the total number of mutants killed by the test suite generated under the corresponding search strategy. Because some programs have a large number of mutants killed while

Strategy	Ave. Kill Rate	Total Kill	Best
$n = 1$	26.00%	14529	14
$n = 2$	27.65%	16072	8
$n = 4$	26.04%	14930	9
$n = 8$	26.96%	13951	8
Combined SGS	**32.17%**	**19719**	**24**
RSS	23.03%	12267	4
DFS	18.63%	9632	2
RPS	25.16%	13349	6
covnew	19.58%	10211	2
depth	21.80%	11105	3
icnt	19.23%	9987	2
md2u	18.44%	9208	2

Table 7. Mutation testing results.

some others only have a few, the total number of killed mutants may skew the contribution from the programs with fewer killed mutants. Thus, we calculate the mutant kill rate for each program and show the average kill rate in column "Ave.Kill Rate". Like what we did in the statement coverage evaluation, we count, for each strategy, its number of the most killed mutants and show the result in column "Best".

Result 6: SGS kills more mutants. From the result, we can see that the subpath-guided search strategies perform better than the other strategies in KLEE. The combined subpath-guided search kills the most mutants and has the highest average kill rate. It should be noted that in this evaluation that the combined search strategy has the best killed number in about 60% of the programs we analyzed, which is significantly lower than the corresponding result from the statement coverage evaluation (around 80%). One possible explanation for this result is that we generated test cases for every terminated execution state, which took more time than only generating test cases covering new statements or branches. In the combined search, each strategy only ran for 15 minutes, and sometimes that is not long enough to explore more parts of the program. Nonetheless, the SGS strategies and its combined version still outperforms all the other strategies.

3.4 Discussions

Our approach provides a general, unified framework to systematically guide symbolic execution using subpath program spectra. We can use a specific n for different purposes. If we want to cover more parts of the program in a short amount of time, a small n (such as 1) can be the desirable choice as it ignores much of the contextual information and can explore program paths with less conditional branches at the very beginning. However, the limited contextual information cannot provide powerful guidance when the coverage reaches a certain level, while some specific parts of the program may need certain loop or iteration times to enter. In this case, a larger n will be a more suitable choice as we can explore the program with more program specific information and thus cover the program more thoroughly. On the other hand, longer n's may also be prone to guide symbolic execution to redundantly explore some paths and reduce the efficiency and effectiveness of path exploration. The combined strategy using different length n appears to be striking a good balance between efficiency and effectiveness.

4. Related Work

We have proposed a unified technique to guide symbolic path exploration to tackle the problem of path explosion, which is a significant challenge for symbolic execution because systematically executing all feasible paths in large programs is costly. This section surveys closely related work on this topic. Many techniques have been proposed in the literature to handle the problem from different aspects: path guidance, path pruning, and parallel execution.

Path Guidance Techniques. Different search strategies have been proposed and used in symbolic or concolic testing. The EXE tool by Cadar *et al.* [8] employs a Best-First Search strategy, which checks all execution states and picks the best one according to some heuristics. Hybrid Concolic Testing [24] is a technique proposed to interleave random testing with concolic execution to obtain a deep and wide exploration. The control-flow guided search strategy [6] constructs a weighted control flow graph (CFG), guiding the exploration to the nearest currently uncovered parts based on the distance in the CFG when the concolic testing needs to choose branches to negate. The fitness-guided search strategy [32] calculates fitness values from explored paths to target predicates and fitness gains for the branches to be flipped, then selects proper paths and branches to cover the target predicates. SAGE [18] proposes a new search algorithm, generational search, to address the practical limitations of path explosion and imperfect symbolic execution. Similar to Best-First Search, generational search tests all children of each expanded execution, scores their entire runs and picks one with the top score to run and check. The search strategy in Pex [30] shares some similarity to ours. It maintains an explored execution tree and picks an outgoing branch every time. However, our strategy is more general and flexible as it provides a spectrum of techniques to guide path exploration. We make decisions guided by more information from the explored paths, which provides better guidance. Our combined strategy also offers a good balance of cost and effectiveness, and can be fruitfully incorporated in existing (directed) symbolic execution engines.

Path Pruning Techniques. There also exist techniques that address the path explosion problem by path pruning. The RWset analysis technique [5] tracks the memory locations read and written by the program to determine whether a path can explore new program behavior. Those execution states that are deemed to produce the same effects as some

already explored paths will be pruned to reduce the number of explored paths. The tool eXpress [29] introduces dynamic symbolic execution for regression test generation and prunes paths that do not expose behavioral differences while exploring new program versions. The SMART [16] technique performs concolic testing compositionally by adapting interprocedural static analysis. While these path pruning techniques focus on avoiding redundant path explorations, our technique focuses on guiding symbolic execution to explore more profitable paths. These are complementary approaches to tackle the path explosion problem and may be fruitfully combined.

Parallel Symbolic Execution. Finally, there are also complementary parallel symbolic execution techniques [28] proposed to mitigate the path explosion problem in symbolic execution. They use simple static partition techniques to help divide the symbolic execution tree and then in parallel explore the partitions utilizing multi-core machines available from cloud or grid computing environments.

5. Conclusion and Future Work

In this paper, we have introduced a general, unified framework to intelligently guide symbolic execution to improve test coverage and error detection. Our key insight is to use length-n subpath program spectra to systematically steer path exploration to less explored parts of a program. We have implemented our framework in a state-of-the-art symbolic execution engine KLEE. Results on a large number of small to medium size real-world programs show that our unified search strategy can generate test cases with higher coverage in less time compared to common traditional strategies. We also show that the generated test cases can help locate more bugs. Finally, we have proposed a natural combination of the specialized strategies under different choices of n and show that it offers the best trade-offs of cost and effectiveness. We believe that our general framework can be incorporated in existing symbolic execution engines besides KLEE (such as Pex and JPF) and have potential applications in other software testing and analysis problems that require path-based analysis.

There are several interesting avenues for future work. First, we would like to explore how to effectively interleave our strategy with other strategies to more intelligently break ties among states with same subpath frequencies. Second, we are also interested in investigating how to dynamically adjust n, for example, by tracking certain program properties. Once the current n cannot provide worthwhile profit for its cost (determined by the properties we track), we may adjust n. Other promising directions include synergistic combinations with path pruning or parallel symbolic execution. Beyond symbolic execution, it would be interesting to apply the same concept to other testing and analysis problems, especially those that rely on path analysis.

Acknowledgments

We thank the anonymous reviewers for constructive feedback on earlier drafts of this paper. This research was supported in part by the National Natural Science Foundation of China (No. 91318301, 61170066, and 61021062), the National 863 High-Tech Program of China (No. 2012AA011205), and United States NSF Grants 0917392, 1117603 and 1319187. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

References

[1] S. Anand, C. S. Păsăreanu, and W. Visser. JPF–SE: A symbolic execution extension to Java PathFinder. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 134–138. Springer, 2007.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Transparent dynamic optimization: The design and implementation of Dynamo. Technical report, Technical Report HPL-1999-78, Hewlett-Packard Laboratories, 1999.

[3] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12. ACM, 2000.

[4] T. Ball and J. Larus. Efficient path profiling. In *ACM/IEEE International Symposium on Microarchitecture*, pages 46–57. IEEE Computer Society, 1996.

[5] P. Boonstoppel, C. Cadar, and D. Engler. RWset: Attacking path explosion in constraint-based test generation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 351–366. Springer, 2008.

[6] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 443–446. IEEE Computer Society, 2008.

[7] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 209–224, 2008.

[8] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):10, 2008.

[9] C. Cadar, P. Godefroid, S. Khurshid, C. Pasareanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: preliminary assessment. In *International Conference on Software Engineering*, pages 1066–1071. IEEE, 2011.

[10] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, 1976.

[11] Coreutils - GNU core utilities. `http://www.gnu.org/software/coreutils/`.

[12] R. DeMillo, R. Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[13] E. Duesterwald and V. Bala. Software profiling for hot path prediction: Less is more. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 202–211. ACM, 2000.

[14] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, 10(4):438–444, 1984.

[15] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(1):63–86, 1996.

[16] P. Godefroid. Compositional dynamic test generation. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 47–54. ACM, 2007.

[17] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 213–223. ACM, 2005.

[18] P. Godefroid, M. Levin, D. Molnar, et al. Automated whitebox fuzz testing. In *Network and Distributed System Security Symposium*. The Internet Society, 2008.

[19] D. Hamlet and R. Taylor. Partition testing does not inspire confidence [program testing]. *IEEE Transactions on Software Engineering*, 16(12):1402–1411, 1990.

[20] M. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing Verification and Reliability*, 10(3):171–194, 2000.

[21] Y. Jia and M. Harman. Milu: A customizable, runtime-optimized higher order mutation testing tool for the full C language. In *Testing: Academia and Industry Conference - Practice And Research Techniques*, pages 94–98. IEEE, 2008.

[22] J. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[23] The KLEE symbolic virtual machine. `http://klee.llvm.org/`.

[24] R. Majumdar and K. Sen. Hybrid concolic testing. In *International Conference on Software Engineering*, pages 416–426. IEEE, 2007.

[25] D. Marinov, A. Andoni, D. Daniliuc, S. Khurshid, and M. Rinard. An evaluation of exhaustive testing for data structures. Technical report, Technical Report MIT-LCS-TR-921, MIT CSAIL, Cambridge, MA, 2003.

[26] C. S. Păsăreanu and W. Visser. Verification of Java programs using symbolic execution and invariant generation. In *Model Checking Software*, pages 164–181. Springer, 2004.

[27] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the Year 2000 problem. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 432–449, 1997.

[28] M. Staats and C. Păsăreanu. Parallel symbolic execution for structural test generation. In *International Symposium on Software Testing and Analysis*, pages 183–194. ACM, 2010.

[29] K. Taneja, T. Xie, N. Tillmann, and J. de Halleux. eXpress: guided path exploration for efficient regression test generation. In *International Symposium on Software Testing and Analysis*, pages 1–11. ACM, 2011.

[30] N. Tillmann and J. De Halleux. Pex–white box test generation for .NET. *Tests and Proofs*, pages 134–153, 2008.

[31] W. Visser, C. S. Păsăreanu, and R. Pelánek. Test input generation for Java containers using state matching. In *International Symposium on Software Testing and Analysis*, pages 37–48. ACM, 2006.

[32] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 359–368. IEEE, 2009.

[33] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 283–294. ACM, 2012.