Scaling Symbolic Execution using Ranged Analysis

Junaid Haroon Siddiqui Sarfraz Khurshid

University of Texas at Austin
1 University Station
Austin, TX 78712
{jsiddiqui,khurshid}@ece.utexas.edu

Abstract

This paper introduces a novel approach to scale symbolic execution—a program analysis technique for systematic exploration of bounded execution paths—for test input generation. While the foundations of symbolic execution were developed over three decades ago, recent years have seen a real resurgence of the technique, specifically for systematic bug finding. However, scaling symbolic execution remains a primary technical challenge due to the inherent complexity of the path-based exploration that lies at core of the technique.

Our key insight is that the *state* of the analysis can be represented highly compactly: a *test input* is all that is needed to effectively encode the state of a symbolic execution run. We present *ranged symbolic execution*, which embodies this insight and uses two test inputs to define a *range*, i.e., the beginning and end, for a symbolic execution run. As an application of our approach, we show how it enables scalability by distributing the path exploration—both in a sequential setting with a single worker node and in a parallel setting with multiple workers. As an enabling technology, we leverage the open-source, state-of-the-art symbolic execution tool KLEE. Experimental results using 71 programs chosen from the widely deployed GNU Coreutils set of Unix utilities show that our approach provides a significant speedup over KLEE. For example, using 10 worker cores, we achieve an average speed-up of 6.6X for the 71 programs.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Symbolic execution

General Terms Algorithms, Performance

Keywords Test input as analysis state, ranged analysis, parallel symbolic execution, incremental analysis, KLEE

1. Introduction

Symbolic execution is a powerful program analysis technique based on a systematic exploration of (bounded) program paths, which was developed over three decades ago [8, 23]. A key idea in symbolic execution is to build *path conditions*—given a path, a path condition represents a constraint on the input variables, which is a conjunction of the branching conditions on the path. Thus, a solution to a (feasible) path condition is an input that executes the corresponding path. A common application of symbolic execution is indeed to generate test inputs, say to increase code coverage. Automation of symbolic execution requires constraint solvers or decision procedures [3, 10] that can handle the classes of constraints in the ensuing path conditions.

A lot of progress has been made during the last decade in constraint solving technology, in particular SAT [39] and SMT [3, 10] solving. Moreover, raw computation power is now able to support the complexity of solving formulas that arise in a number of real applications. These technological advances have fueled the research interest in symbolic execution, which today not only handles constructs of modern programming languages and enables traditional analyses, such as test input generation [7, 15, 22, 33], but also has non-conventional applications, for example in checking program equivalence [31], in repairing data structures for error recovery [12], and in estimating power consumption [34].

Despite the advances, a key limiting factor of symbolic execution remains its inherently complex path-based analysis. Several recent research projects have attempted to address this basic limitation by devising novel techniques, including compositional [14], incremental [30], and parallel [6, 16, 36, 40] techniques. While each of these techniques offers its benefits (Section 5), a basic property of existing techniques is the need to apply them to completion in a single execution if *completeness* of analysis (i.e., complete exploration of the bounded space of paths) is desired. Thus, for example, if a technique times out, we must re-apply it for a greater time bound, which can represent a costly waste of computations that were performed before the technique timed out.

This paper presents *ranged symbolic execution*, a novel technique for scaling symbolic execution for test input generation. Our key insight is that the *state* of a symbolic execution run can, rather surprisingly, be encoded succinctly by a *test input*—specifically, by the input that executes the last terminating (feasible) path explored by symbolic execution. By defining a fixed *branch exploration ordering*—e.g., taking the true branch before taking the false branch at each non-deterministic branch point during the exploration—an operation already fixed by common implementations of symbolic execution [2, 7, 22], we have that each test input partitions the space of (bounded) paths under symbolic execution into two sets: *explored* paths and *unexplored* paths. Moreover, the branch exploration ordering defines a *linear order* among test inputs; specifically, for any two inputs (that do not execute the same path or lead to an infinite loop), the branching structure of the corresponding paths defines which of the two paths will be explored first by symbolic execution. Thus, an ordered pair of tests, say $\langle \tau, \tau' \rangle$, defines a *range* of (bounded) paths $[\rho_1, \ldots, \rho_k]$ where path ρ_1 is executed by τ and path ρ_k is executed by τ', and for $1 \leq i < k$, path ρ_{i+1} is explored immediately after path ρ_i.

Encoding the analysis state as a test input has a number of applications. The most direct one is to enable symbolic execution to be *paused* and *resumed*. To illustrate, if an analysis runs out of resources, the last test input generated allows it to be effectively paused for resumption later (possibly on another machine with greater resources) without requiring the previously completed work to be re-done. Another key application, which is the focus of this paper, is a novel way to partition the path exploration in symbolic execution to scale it—both in a sequential setting with one worker node and in a parallel setting with several workers. The encoding allows dividing the problem of symbolic execution into several sub-problems of ranged symbolic execution, which have minimal overlap and can be solved separately. It also allows effective load balancing in a parallel setting using dynamic refinement of ranges based on work stealing with minimal overhead due to the compactness of a test input.

We make the following contributions:

- **Test input as analysis state.** We introduce the idea of encoding the state of a symbolic execution run using a single test input.
- **Resumable symbolic execution.** Our encoding allows symbolic execution to be paused and resumed using minimal book-keeping—just a single test input.
- **Two test inputs as analysis range.** We introduce the idea of using two test inputs to define a range of paths to be explored using symbolic execution and to restrict it to that range.
- **Ranged symbolic execution.** Restricting symbolic execution to a range allows simply using a set of inputs to divide the problem of symbolic execution of all bounded execution paths into a number of sub-problems of ranged symbolic execution, which can be solved separately.
- **Dynamic range refinement using work stealing.** We introduce load-balancing for parallel symbolic execution using dynamically defined ranges that are refined using work stealing.
- **Implementation.** We implemented ranged symbolic execution using KLEE [7]—an open-source symbolic execution tool, which analyzes LLVM [1], an intermediate compiler language that is closer to assembly and only has two-way branches, but has more type/dependency information than assembly. We developed a work stealing version using MPI [38] message communication.
- **Evaluation.** We evaluated ranged symbolic execution using 71 programs from GNU Coreutils—the widely deployed set of Unix utilities. We observed an average speedup of 6.6X for the 71 programs using 10 workers.

2. Illustrative overview

Forward symbolic execution is a technique for executing a program on symbolic values [9, 23]. There are two fundamental aspects of symbolic execution: (1) defining semantics of operations that are originally defined for concrete values and (2) maintaining a *path condition* for the current program path being executed – a path condition specifies necessary constraints on input variables that must be satisfied to execute the corresponding path.

As an example, consider the following program that returns the middle of three integer values.

```
1  int mid(int x, int y, int z) {
2    if (x<y) {
3      if (y<z) return y;
4      else if (x<z) return z;
5      else return x;
6    } else if (x<z) return x;
7    else if (y<z) return z;
8    else return y; }
```

To symbolically execute this program, we consider its behavior on integer inputs, say X, Y, and Z. We make no assumptions about the value of these variables (except what can be deduced from the type declaration). So, when we encounter a conditional statement, we consider both possible outcomes of the condition. We perform operations on symbols algebraically.

Symbolic execution of the program `mid` explores 6 paths:

```
path 1: [X < Y < Z] L2 -> L3
path 2: [X < Z < Y] L2 -> L3 -> L4
path 3: [Z < X < Y] L2 -> L3 -> L4 -> L5
path 4: [Y < X < Z] L2 -> L6
path 5: [Y < Z < X] L2 -> L6 -> L7
path 6: [Z < Y < X] L2 -> L6 -> L7 -> L8
```

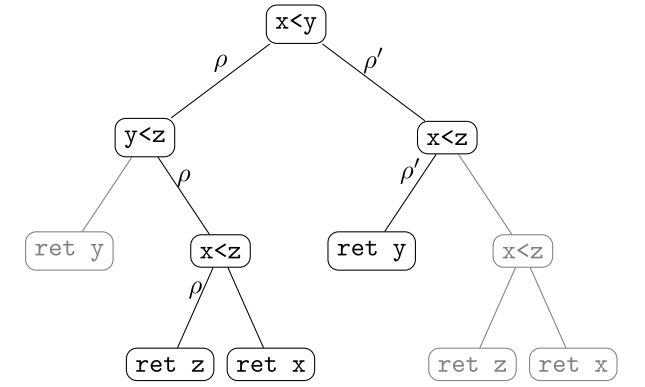


Figure 1. Symbolic execution between paths ρ and ρ'.

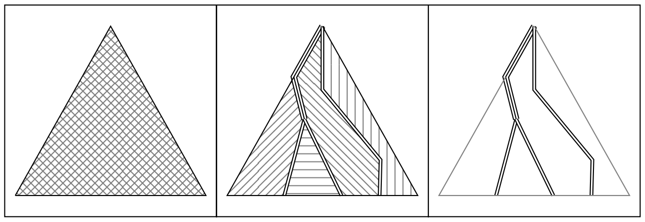
Note that for each path that is explored, there is a corresponding path condition (shown in square brackets). While execution on a concrete input would have followed exactly one of these paths, symbolic execution explores all six paths.

The path conditions for each of these paths can be solved using off-the-shelf SAT solvers for concrete tests that exercise the particular path. For example, path 2 can be solved to `X=1`, `Y=3`, and `Z=2`.

Ranged symbolic execution enables symbolic exploration between two given paths. For example, if path 2 and path 4 are given, it can explore paths 2, 3, and 4. In fact, it only needs the concrete solution that satisfies the corresponding path condition. Therefore it is efficient to store and pass paths. Ranged symbolic execution builds on a number of key observations we make:

- A concrete solution corresponds to exactly one path in code and it can be used to re-build the path condition that leads to that path. Solving a path condition to find concrete inputs is computationally intensive. However, checking if a given solution satisfies a set of constraints is very light-weight. Thus we can symbolically execute the method again and at every branch only choose the direction satisfied by the concrete test.
- We can define an ordering on all paths if the true side of every branch is always considered before the false side. Since, every concrete test can be converted to a path, the ordering can be defined over any set of concrete inputs.
- Using two concrete inputs, we can find two paths in the program and we can restrict symbolic execution between these paths according to the ordering defined above. We call this *ranged symbolic execution*.

For example, consider that we are given test inputs `τ(X=1, Y=3, Z=2)` and `τ′(X=2, Y=1, Z=3)` which take paths ρ and ρ' in code (path 2 and path 4 in above example), and we want to symbolically execute the range between them. We show this example in Figure 1. We start symbolic execution as normal and at the first comparison `x<y`, we note that ρ goes to the `true` side while ρ' goes to the `false` side.



(a) Standard symbolic execution

(b) Ranged symbolic execution (4 ranges)

(c) Only boundaries redundantly analyzed

Figure 2. High level overview of dividing symbolic execution into non-overlapping ranges for independent symbolic executions.

At this point, we also know that $\rho < \rho'$ in the ordering we defined. Thus, when $x < y$, we only explore what comes after ρ in the ordering and when $x \not< y$ we explore what comes before ρ'. At the next comparison `y<z` we skip the `true` side and only explore the `false` side satisfied by ρ. Similarly we can skip three states using ρ'. Skipped states are drawn in gray color in Figure 1. Three paths are explored as a result. We consider the range $[\tau, \tau')$ as a half-open interval where the start is considered part of the range but not the end. Thus we produce two test cases as a result.

Once we have the basic mechanism for ranged symbolic execution, we use it in three novel ways:

- **Resumable execution:** Ranged symbolic execution enables symbolic execution to be paused at any stage and it can be restarted later with minimal overhead using the last input it generated as the start of new range.
- **Parallel execution:** Ranges of symbolic execution can be analyzed in parallel. For example, we can have three *non-overlapping* ranges for the above example `[null,τ)`, `[τ,τ′)`, and `[τ′,null)`, where `null` designates unbounded end of a range. Executing these in parallel will completely analyze the above function without *any* communication between parallel nodes. Figure 2 shows a high level overview of dividing symbolic execution into non-overlapping ranges. Only the paths dividing the ranges are redundantly analyzed as path of both ranges. The initial set of dividing points can come from manual or random test cases or from symbolic execution of a previous version of code.
- **Dynamic range refinement:** We further provide an algorithm for dynamic range refinement that enables parallel symbolic execution using work stealing when there is no initial set of inputs to form the ranges. For example, if a parallel node starts symbolic execution of the `mid` function and reaches the first branch `x<y`, it proceeds with the `true` branch while queueing the `false` branch in a list of work to be finished later. In the meanwhile, if another parallel node is free for work, it can steal work from the queue of this node and explore paths where `!(x<y)`.

3. Technique

In this section, we discuss using a single test case as analysis state and using two test cases to define an analysis range (Section 3.1), performing symbolic execution within a range (Section 3.2), using ranged symbolic execution for parallel and resumable analysis (Section 3.3), and dynamic range refinement to enable distributed work stealing (Section 3.4). Our presentation assumes a standard bounded depth-first symbolic execution where path exploration is systematic and for any condition, the "true" branch is explored before the "false" branch. Such exploration is standard in commonly used tools such as KLEE [7], JPF [26], and CUTE [33].

3.1 Test input as analysis state

We introduce three concepts in this section: (1) describing analysis state with a single concrete test, (2) defining ordering of tests based on paths taken, and (3) using two concrete tests to define a range of analysis.

We introduce the concept of describing analysis state with a single concrete test.

Definition 1. *Given the total order $\mathcal{O}$ of all paths ρ taken by a path-based analysis, any concrete test τ defines a state of analysis in which every path $\rho < \rho_\tau$ under $\mathcal{O}$ has been explored and none of the rest has been explored. ρ_τ is the path taken by test τ.*

Definition 1 assumes the existence of translation from concrete tests to paths and an algorithm to compare tests based on the ordering taken by the path-based analysis. The translation from concrete tests to paths can be done simply by executing the test and observing the path it takes. In practice, however, we will not need to find the corresponding path separately and it will be calculated along with other operations as discussed in the next section. Next, we discuss *test ordering*.

Definition 2. *Given two tests τ and τ' and the corresponding paths ρ and ρ', where $\langle \flat_0, \ldots, \flat_i \rangle$ is the set of basic blocks in ρ and $\langle \flat'_0, \ldots, \flat'_i \rangle$ is the set of basic blocks in ρ', we define that $\rho < \rho'$ if and only if there exists a k such that $\forall_{i=0}^{k} \flat_i = \flat'_i$ and the terminating instruction in $\flat_k$ is a conditional branch with $\flat_{k+1}$ as the "then" basic block and $\flat'_{k+1}$ as the "else" basic block.*

Definition 2 orders tests based on the paths they take. We find the first branch where the two paths differ. We consider the test taking the "true" branch *smaller* than the test taking the "false" branch. If two tests take the same path till the end, we consider them *equivalent*. Ordering more than two tests can be done by any sorting algorithm.

Definition 3. *Let τ and τ' be two tests with execution paths ρ and ρ' respectively, we define a range $[\tau, \tau')$ to be the set of all paths ρ_i such that $\rho \leq \rho_i < \rho'$.*

Algorithm 1: Algorithm to compare two tests. This can be used with any sorting algorithm to order any number of tests.

```
input  : test $\tau$, test $\tau'$
output: BIGGER, SMALLER, or EQUIVALENT
1  define path-cond $\rho$, address-space AS, address-space AS';
2  $i$ = first instruction in func;
3  repeat
4      if $i$ is-a conditional branch then
5          cond ← condition of $i$;
6          if PathTakenByTest($\tau$, $\rho \wedge$cond, AS) then
7              if NOT PathTakenByTest($\tau'$, $\rho \wedge$cond, AS') then
8                  return BIGGER;
9              end
10             $\rho \leftarrow \rho \wedge$ cond;
11             $i \leftarrow$ first instruction in then basic block;
12         else
13             if PathTakenByTest($\tau'$, $\rho \wedge$cond, AS') then
14                 return SMALLER;
15             end
16             $\rho \leftarrow \rho \wedge$ NOT(cond);
17             $i \leftarrow$ first instruction in else basic block;
18         end
19     else
20         update AS for $i$ using $\tau$;
21         update AS' for $i$ using $\tau'$;
22         $i \leftarrow$ successor of $i$;
23     end
24 until $i$ is the last instruction;
25 return EQUIVALENT;
```

The benefit of defining a half-open range is that given three tests $\tau_a < \tau_b < \tau_c$, we have $[\tau_a, \tau_c) = [\tau_a, \tau_b) + [\tau_b, \tau_c)$.

We extend this concept to a set of n tests. We can find the paths taken by these tests and order them using the above algorithm. If the tests take p distinct paths ($p \leq n$), they define $p + 1$ ranges of paths. Note that $p < n$ when multiple tests take the same path in code and are thus *equivalent*. The first and last range use special tests `begin` and `end`, where `begin` is the *smallest* path and `end` is one beyond the *biggest* path. The `end` is defined as one beyond the last path because we define ranges as half-open and we want all paths explored.

Lemma 1. *Ranged analyses on a set of $n - 1$ ranges $[\tau_1, \tau_2)$, ..., $[\tau_{n-1}, \tau_n)$ explore the same set of paths as the ranged analysis on $[\tau_1, \tau_n)$.*

This paper presents algorithms to perform symbolic execution of a program using ranges defined by tests. It shows how to *efficiently* perform symbolic execution of only the paths within a range. These algorithms are presented in the next subsection.

3.2 Ranged symbolic execution

In this section, we apply the technique of defining ranges of path-based analysis to symbolic execution. We call this *ranged symbolic execution*.

Definition 4. *Let τ and τ' be two tests that execute paths ρ and ρ' where $\rho < \rho'$. Define ranged symbolic execution for $[\tau, \tau')$ as symbolic execution of all paths ρ_i such that $\rho \leq \rho_i < \rho'$.*

Performing ranged symbolic execution has two parts: (1) defining the ranges given a set of tests and (2) symbolically executing the paths within ranges.

To define ranges given a set of tests, we can use any sorting algorithm given a comparator to compare two tests. Two tests can be compared either by running them independently and analyzing the branches they take or we can analyze two paths simultaneously until we find the first difference. The benefit of the second technique is that we only need to execute the common part of two paths and not explore two complete paths.

Algorithm 1 gives the algorithm for analyzing the common part of paths taken by two tests. The algorithm depends on a predicate function that checks if a given test satisfies a given condition. For that, we symbolically evaluate the path condition for the values in the given test. Note that checking if a path condition is satisfied by a given input is a very efficient operation. In contrast, we need much more time to solve a path condition to generate concrete test inputs using a SAT solver. The two tasks differ in complexity.

To run symbolic execution *between* two tests, we have to (1) convert them into paths, (2) find all paths between them, and (3) execute those paths symbolically. We do all three tasks simultaneously and thus we have no intermediate storage requirements.

Symbolic execution state for a particular path contains the set of path constraints and address space. At branches, the state is split into two states. States to be visited in the future are added to a queue of states. The order of choosing states from the queue determines the search strategy used. We use depth first search in this work.

For restricting symbolic execution to a range, we introduce new variables to represent the starting and ending tests in the symbolic execution state. The initial state gets the starting and ending test parameters from program input. If one of the parameters is the special *begin* or *end* symbol (i.e. its unbounded), we just use `null` in its place. We perform symbolic execution normally while using the function in Algorithm 2 for conditional branches.

Algorithm 2: Algorithm for handling a branch for ranged symbolic execution. Each state works within a range defined by a start test τ_{start} and an end test τ_{end}. A new state is created using a basic block to start execution from, and a pair of tests to define the range.

```
input  : state, branch, test $\tau_{start}$, test $\tau_{end}$
output: set of states to be explored
1  cond ← branch condition of branch;
2  $BB_{then}$ ← then basic block of branch;
3  $BB_{else}$ ← else basic block of branch;
4  if $\tau_{start} \wedge \neg(\tau_{start} \Rightarrow cond)$ then
5      return {new state($BB_{else}$, $\tau_{start}$, $\tau_{end}$)};
6  end
7  if $\tau_{end} \wedge \tau_{end} \Rightarrow cond$ then
8      return {new state($BB_{then}$, $\tau_{start}$, $\tau_{end}$)};
9  end
10 if cond is unsatisfiable then
11     return {new state($BB_{else}$, $\tau_{start}$, $\tau_{end}$)};
12 else if ¬cond is unsatisfiable then
13     return {new state($BB_{then}$, $\tau_{start}$, $\tau_{end}$)};
14 else if both are unsatisfiable then
15     // triggers for unreachable code;
16     return $\emptyset$;
17 else
18     return {new state($BB_{then}$, $\tau_{start}$, null), new
           state($BB_{else}$, null, $\tau_{end}$) };
19 end
```

Algorithm 2 works by checking if the current state has a starting test assigned *and* that starting test does not satisfy the branch condition. Since we defined test ordering with *true* branches preceding *false* branches, we need to eliminate the *true* branch from the search. Similarly if we have an ending test which *does* satisfy the branch condition, we eliminate the *false* side from being explored.

3.3 Parallel and resumable analysis

Ranged symbolic execution enables parallel and resumable analysis. For parallel analysis, we take a set of tests and use them to divide the symbolic analysis into a number of ranges. These ranges are then evaluated in parallel. We can use more ranges then available workers so that workers that finish quickly can pick another range from the work queue. The initial set of tests can come from manual tests, a symbolic execution run on a previous version of code, or even from a shallow symbolic execution run on the same code. In our evaluation, we pick random collection of tests from a sequential run and use it to define ranges for the parallel run. In the next section, we introduce another way to parallelize that requires no initial set of tests. It uses work stealing to get to-be-explored states from a busy worker to a free worker, and in doing so, dynamically redefining the ranges for both workers.

Ranged symbolic execution also enables resumable execution, where we can pause symbolic execution and resume it by giving it a concrete test corresponding to the last path explored as the starting point. To use it in combination with parallel analysis, we would also need the original ending point of the paused range. In the evaluation, we show a scheme, where pre-defined ranges are analyzed in increments resulting in negligible overhead and greater flexibility.

3.4 Dynamic range refinement

Dynamic range refinement enables dynamic load balancing for ranged symbolic execution using work with work stealing. It starts with a single worker node responsible for the complete range $[a, c)$. Whenever this node hits branches it explores the *true* side and puts the *false* side on a queue to be considered later. As other workers come, they can steal work from this queue. The state on the queue is persisted as a test case b and the range is redefined to $[a, b)$. The stolen range $[b, c)$ is taken up by another worker.

Our implementation of distributed symbolic execution using work stealing utilizes a master coordinator node and uses MPI for communication. Algorithm 3 gives the algorithm for work stealing coordinator. It maintains lists for waiting workers and busy workers. Whenever a node needs work it tries to find a busy worker and tries to steal work. If a previously started stolen work request completes, it passes the work to a waiting worker. Sometimes, a stolen work request fails because the node is already finished or there is no work in the queue at that time. In such a case, it tries to steal work from another worker node.

Algorithm 4 is the algorithm for a worker node. When it receives a range from the coordinator, it performs ranged symbolic execution on it. If it receives a request to steal work, it checks if there is any state in the work queue. If so, it converts the request to a concrete test to easily pass to the coordinator, and redefines the current symbolic execution range to end at that test. If there is no state in the work queue, it informs the coordinator of a failure. The worker repeats getting work and stealing ranges until the coordinator tells it to shut down.

Using intermediate states in this manner is different from using concrete tests that represent complete paths in code (like Section 3.3). Intermediate states, on the other hand, represent partial paths. Partial paths can result in overlapping ranges and more work that absolutely necessary. We circumvent this by choosing zero values for any fields not accessed by the concrete test. This extends the partial path to make a complete path that satisfies a zero value assignment for the remaining fields. It is possible that such a path ends up being infeasible, but it is a complete path and sufficient to define non-overlapping symbolic execution ranges.

Algorithm 3: Algorithm for work stealing coordinator.

```
1  define lists of waiting workers and busy workers;
2  count of workers with no theft started = 0;
3  give the whole task to the first worker;
4  while true do
5      receive message $m$ from worker $w$;
6      if $m$=need work then
7          find a worker $w_2$ where no theft has been initiated;
8          if no such process then
9              increment count of workers with no theft started;
10             if this count = total number of workers then
11                 terminate, we are done;
12             end
13         else
14             ask $w_2$ to give stolen work;
15         end
16         add $w$ to list of waiting workers;
17     else if $m$=stolen work then
18         give stolen work to a waiting worker $w_2$;
19         remove $w_2$ from list of waiting workers;
20         if count of workers with no theft started $> 0$ then
21             ask $w_2$ to give stolen work (again);
22             decrement count of workers with no theft started;
23         end
24     else if $m$=cannot steal then
25         choose another busy worker $w_2$;
26         ask $w_2$ to give some stolen work;
27     end
28 end
```

4. Evaluation

To evaluate ranged symbolic execution, we consider the following research questions:

- How does ranged symbolic execution perform in a sequential setting with respect to standard symbolic execution?
- How does ranged symbolic execution perform in a parallel setting using statically defined ranges with respect to standard symbolic execution?
- How does ranged symbolic execution in a parallel setting using dynamic range refinement perform in comparison with using statically defined ranges?
- How does ranged symbolic execution in a parallel setting using dynamic range refinement scale?

Algorithm 4: Algorithm for work stealing worker node performing ranged symbolic execution.

```
while true do
    receive message $m$ from coordinator;
    if $m$=exit then
        terminate;
    end
    else if $m$=new work then
        start ranged symbolic execution of new work ;
    else if $m$=steal work then
        if stealable states exist in symbolic execution state then
            remove state and convert it to a concrete test;
            send the concrete test to coordinator;
            update the end of current symbolic execution range;
        else
            inform coordinator that stealing failed
        end
    end
end
```

In the following subsections, we describe (1) the set of test programs we use, (2) our methodology, (3) the experimental results, and (4) the threats to validity.

4.1 Subjects

To evaluate ranged symbolic execution, we use GNU core utilities (Coreutils)[1] — the basic file, shell, and text manipulation core utilities for the GNU operating system. Coreutils are medium sized programs between 2000 and 6000 lines of code. Some of these programs do a particular task with a lot of error checks and thus form a deep search tree while others perform multiple functions and form a broad search tree. Deep trees likely provide less opportunity for efficient parallel analysis than broad trees. These utilities provide a good mix of subject programs where parallelism in symbolic execution likely helps for some but not others.

Coreutils were also used in the evaluation of the KLEE symbolic execution tool [7]. As, we build ranged symbolic execution using KLEE, Coreutils provide a good benchmark for comparison with KLEE.

We ran KLEE on each program in Coreutils for ten minutes and chose the 71 utilities for which KLEE covered more than a hundred paths in this time.

4.2 Methodology

In this section, we discuss our evaluation setup, how we ensure that all techniques cover the same paths for a fair comparison, how we define static ranges, and how we setup work stealing.

We performed the experiments on the Lonestar Linux cluster at the Texas Advanced Computing Center (TACC)[2]. TACC enables reliable experiments as the processors are fully allocated to one job at a time.

Ranged symbolic execution and standard symbolic execution cover the same paths under a given depth bound. However, our experiments also use a time bound of 10 minutes. Since ranged symbolic execution analyzes paths in parallel starting from many starting points, the paths it covers in 10 minutes may not be the same as those covered by standard symbolic execution on the same program in 10 minutes. To allow fair comparison we use the last test generated by standard symbolic execution as an upper bound for the ranged executions. Thus, we ensure that every technique covers the same paths. The time of standard symbolic execution shown in the tables is calculated from the start of execution to when this last completed path was covered.

For evaluating the performance of ranged symbolic execution using static ranges, we choose nine tests at random from those generated using standard symbolic execution to define ten ranges for ranged symbolic execution. The end of the last range is fixed to the last test generated by standard symbolic execution (as discussed above). As the performance of ranged symbolic execution depends on the tests chosen randomly, we repeat the random selection and ranged symbolic execution five times and find the minimum, maximum, and average of the times taken. We also find the minimum, maximum, and average times for the range taking the longest time for each set.

For evaluating the performance of ranged symbolic execution using dynamic range refinement, we use 10 worker processors and 1 coordinator processor to symbolically execute the same problem with no a priori division. This experiment is not repeated multiple times as there is no nondeterministic choice of ranges to be made. All ranges are dynamically formed.

For evaluating how ranged symbolic execution in a parallel setting using dynamic range refinement scales, we choose 15 programs and run parallel symbolic execution using 5, 10, and 20 workers. Specifically, we choose 5 programs that gave the worst speedup, 5 programs that gave the median speedup, and 5 programs that gave the best speedup using dynamic range refinement on 10 workers.

4.3 Experimental results

Table 1 shows the results for all 71 programs we tested. The second column has time for standard symbolic execution using KLEE. The third column gives the minimum, maximum, and average times for covering the same paths divided into 10 ranges at random. The fourth column has the minimum, maximum, and average time for the range taking the most

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

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

	Standard	Resumable sym-	Parallel symbolic execution using 10 workers			
	symbolic	bolic execution	using static random ranges		using work	
Program	execution	time (s)	time (s)		stealing	
Name	time (s)	min / avg / max	min / avg / max	speedup	time (s)	speedup
base64	600	365 / 377 / 388	68 / 100 / 119	5.0 - 8.8X	83	7.2X
basename	156	110 / 115 / 126	18 / 32 / 63	2.5 - 8.6X	47	3.3X
cat	600	465 / 497 / 518	114 / 175 / 247	2.4 - 5.3X	90	6.6X
chcon	596	401 / 438 / 479	233 / 251 / 283	2.1 - 2.6X	193	3.1X
chgrp	569	283 / 301 / 325	68 / 138 / 175	3.3 - 8.4X	41	13.8X
chmod	550	243 / 256 / 267	73 / 78 / 88	6.2 - 7.6X	46	12.0X
chown	598	263 / 283 / 300	64 / 87 / 120	5.0 - 9.4X	41	14.4X
chroot	599	358 / 393 / 414	102 / 151 / 238	2.5 - 5.8X	330	1.8X
comm	607	730 / 929 / 1125	338 / 472 / 599	1.0 - 1.8X	630	1.0X
cp	600	231 / 264 / 290	58 / 120 / 175	3.4 - 10.3X	56	10.8X
csplit	601	349 / 366 / 387	105 / 162 / 196	3.1 - 5.7X	57	10.5X
cut	600	427 / 442 / 465	144 / 171 / 221	2.7 - 4.2X	105	5.7X
date	278	252 / 260 / 275	83 / 113 / 130	2.1 - 3.3X	84	3.3X
dd	601	353 / 379 / 402	121 / 162 / 195	3.1 - 5.0X	278	2.2X
df	341	151 / 153 / 154	38 / 59 / 69	5.0 - 8.9X	49	7.0X
dircolors	600	460 / 468 / 485	101 / 147 / 198	3.0 - 5.9X	113	5.3X
dirname	618	628 / 701 / 758	377 / 534 / 616	1.0 - 1.6X	574	1.1X
du	601	482 / 540 / 578	134 / 180 / 232	2.6 - 4.5X	115	5.2X
echo	600	400 / 419 / 441	112 / 156 / 203	3.0 - 5.3X	101	6.0X
env	600	492 / 503 / 512	114 / 171 / 236	2.5 - 5.3X	116	5.2X
expand	600	334 / 352 / 367	60 / 110 / 169	3.6 - 10.1X	59	10.2X
factor	609	609 / 622 / 640	93 / 156 / 185	3.3 - 6.5X	540	1.1X
fmt	601	743 / 781 / 826	142 / 176 / 215	2.8 - 4.2X	255	2.4X
fold	600	216 / 227 / 246	62 / 73 / 83	7.3 - 9.7X	45	13.3X
ginstall	596	429 / 451 / 500	105 / 163 / 232	2.6 - 5.7X	281	2.1X
groups	588	658 / 667 / 686	130 / 169 / 214	2.7 - 4.5X	350	1.7X
head	600	229 / 282 / 380	42 / 111 / 246	2.4 - 14.3X	85	7.1X
id	600	257 / 270 / 293	104 / 125 / 140	4.3 - 5.7X	49	12.3X
join	594	499 / 530 / 582	108 / 131 / 162	3.7 - 5.5X	192	3.1X
kill	600	207 / 214 / 223	43 / 65 / 107	5.6 - 13.9X	76	7.9X
ln	600	179 / 213 / 255	38 / 99 / 166	3.6 - 16.0X	53	11.4X
mkdir	596	605 / 735 / 847	259 / 313 / 400	1.5 - 2.3X	474	1.3X
mknod	609	549 / 790 / 1134	485 / 555 / 662	0.9 - 1.3X	572	1.1X
mktemp	600	352 / 375 / 402	197 / 212 / 256	2.3 - 3.1X	240	2.5X
mv	598	438 / 482 / 601	257 / 305 / 335	1.8 - 2.3X	353	1.7X
nice	600	254 / 299 / 368	80 / 153 / 255	2.3 - 7.5X	64	9.4X
nl	600	253 / 285 / 330	72 / 141 / 210	2.9 - 8.3X	53	11.3X
nohup	601	323 / 365 / 422	107 / 185 / 276	2.2 - 5.6X	290	2.1X
od	601	609 / 637 / 654	120 / 209 / 264	2.3 - 5.0X	122	4.9X
paste	600	380 / 397 / 433	85 / 130 / 206	2.9 - 7.1X	83	7.3X
pathchk	599	313 / 364 / 442	100 / 169 / 208	2.9 - 6.0X	178	3.4X
pinky	600	173 / 198 / 227	47 / 67 / 81	7.4 - 12.7X	46	13.1X
pr	601	538 / 580 / 606	93 / 169 / 237	2.5 - 6.4X	108	5.6X
printenv	588	337 / 549 / 749	96 / 251 / 352	1.7 - 6.2X	46	12.8X
printf	598	188 / 219 / 273	53 / 81 / 121	4.9 - 11.3X	46	12.9X
readlink	600	247 / 266 / 305	86 / 108 / 137	4.4 - 7.0X	41	14.7X
rm	603	344 / 375 / 392	109 / 148 / 194	3.1 - 5.6X	185	3.3X
runcon	598	227 / 252 / 280	54 / 86 / 141	4.2 - 11.2X	55	10.8X
seq	600	287 / 312 / 333	90 / 110 / 133	4.5 - 6.6X	105	5.7X
setuidgid	600	507 / 552 / 623	95 / 156 / 206	2.9 - 6.3X	253	2.4X
sha1sum	600	312 / 324 / 332	72 / 111 / 144	4.2 - 8.4X	70	8.6X
shred	600	334 / 397 / 452	96 / 154 / 203	2.9 - 6.3X	95	6.3X
shuf	600	338 / 358 / 380	82 / 114 / 142	4.2 - 7.3X	74	8.1X
split	600	496 / 513 / 524	134 / 206 / 254	2.4 - 4.5X	123	4.9X
stat	599	246 / 268 / 290	73 / 88 / 104	5.8 - 8.2X	79	7.6X
stty	601	154 / 170 / 183	37 / 49 / 74	8.2 - 16.5X	63	9.6X
su	418	331 / 340 / 348	115 / 134 / 143	2.9 - 3.6X	300	1.4X
sum	600	240 / 282 / 340	86 / 136 / 204	2.9 - 7.0X	52	11.5X
tac	602	381 / 480 / 579	210 / 313 / 406	1.5 - 2.9X	160	3.8X
tail	600	349 / 369 / 397	102 / 152 / 204	2.9 - 5.9X	81	7.4X
tee	600	280 / 306 / 336	84 / 128 / 207	2.9 - 7.1X	50	12.0X
touch	561	312 / 333 / 371	81 / 115 / 157	3.6 - 7.0X	282	2.0X
tr	597	497 / 638 / 730	395 / 459 / 583	1.0 - 1.5X	569	1.0X
tsort	600	541 / 545 / 551	113 / 153 / 189	3.2 - 5.3X	121	5.0X
tty	588	517 / 530 / 556	174 / 222 / 308	1.9 - 3.4X	294	2.0X
uname	599	156 / 194 / 230	31 / 71 / 109	5.5 - 19.3X	34	17.7X
unexpand	600	508 / 528 / 541	102 / 148 / 196	3.1 - 5.9X	121	5.0X
uniq	600	370 / 391 / 430	119 / 150 / 175	3.4 - 5.0X	58	10.3X
vdir	596	377 / 440 / 553	162 / 263 / 425	1.4 - 3.7X	125	4.8X
wc	600	555 / 570 / 591	109 / 136 / 187	3.2 - 5.5X	125	4.8X
who	600	304 / 332 / 377	69 / 123 / 225	2.7 - 8.8X	70	8.6X
Average	581	371 / 409 / 454	117 / 166 / 220	3.3 - 6.8X	160	6.6X

Table 1. Ranged symbolic execution for resumable and parallel checking for 71 program from GNU Coreutils suite of Unix utilities. At times the speedup is greater than 10X because of optimal use of caches in KLEE. KLEE is likely more efficient at solving multiple smaller problems than a single large problem.

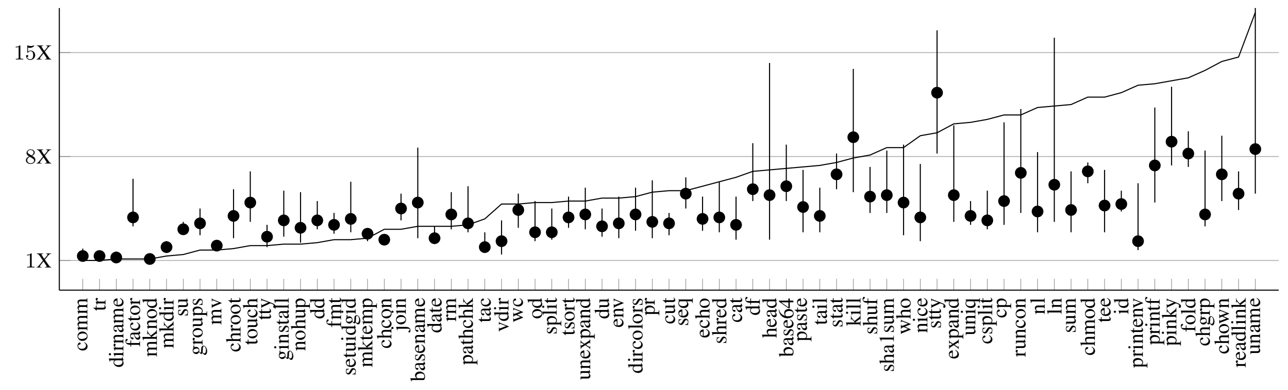


Figure 3. Speedup with 10 worker nodes using ranged symbolic execution for 71 program from GNU Coreutils suite of Unix utilities. Vertical bars show the range of speedup achieved using different random static ranges with the average pointed out. The line shows the speedup achieved using dynamic load balancing using work stealing.

Program	Serial	5+1p		10+1p		20+1p	
Name	time(s)	time(s)	speedup	time(s)	speedup	time(s)	speedup
comm	607	573	1.1X	630	1.0X	514	1.2X
tr	597	509	1.2X	569	1.0X	595	1.0X
dirname	618	567	1.1X	574	1.1X	526	1.2X
factor	609	557	1.1X	540	1.1X	482	1.3X
mknod	609	593	1.0X	572	1.1X	505	1.2X
dircolors	600	142	4.2X	113	5.3X	95	6.3X
pr	601	138	4.4X	108	5.6X	86	7.0X
cut	600	130	4.6X	105	5.7X	67	9.0X
seq	600	129	4.7X	105	5.7X	102	5.9X
echo	600	139	4.3X	101	6.0X	38	15.8X
fold	600	62	9.7X	45	13.3X	32	18.8X
chgrp	569	74	7.7X	41	13.8X	39	14.6X
chown	598	68	8.8X	41	14.4X	39	15.3X
readlink	600	63	9.5X	41	14.7X	34	17.6X
uname	599	48	12.5X	34	17.7X	27	22.2X
Average	600.5	252.8	5.1X	241.3	7.2X	212.1	9.2X

Table 2. Ranged symbolic execution with work stealing for 15 programs from GNU Coreutils on different number of workers. The +1 designates a separate coordinator node. These are the worst 5, median 5, and best 5 utilities from Figure 3 based on performance on 10 workers.

time using the same ranges. Note that while the total time is pretty close for different random ranges, the time for the range taking the most time varies a lot. Thus, the benefit of running in parallel depends on how good a static range is. This restriction applies to other parallel schemes as well that use static partitioning, e.g. [40]. The next column shows the calculated range of speedup achieved. The last two columns have the time and speedup for 11 processors (10 workers and 1 coordinator) when performing parallel symbolic execution using work stealing. We chose 10 workers so that the times can be directly compared to the times for 10 parallel workers using random static ranges (column 4).

Speedup for parallel symbolic execution using work stealing ranges from 1.0X (no speedup) to 17.7X. As 17.7 is more than the number of workers, we investigated and found that KLEE uses a lot of internal caches which can perform much better when they are of a smaller size. Thus, KLEE is likely more efficient at solving smaller problems than one big problem. This is intuitive as symbolic execution maintains a lot of internal state and memory maps with frequent search operations. These search operations become more efficient for smaller problems (with or without caching). Thus, ranged symbolic execution often makes KLEE faster even when all ranges are executed sequentially. We also note that 13 of the 71 utilities observed a slowdown in at least one run in a resumable setting. However, on average (last row in Table 1), even the worst resumable run is faster than a standard execution. Thus, in most cases, we observe better performance with resumable symbolic execution.

Figure 3 contains a plot of the speedup of all 71 utilities ordered by the speedup achieved using work stealing. The line graph shows the speedup for parallel symbolic execution using work stealing, while the vertical lines show the range of speedup for parallel symbolic execution using static random ranges. The dot on the vertical line shows the average speedup for static ranges. Note that for a third of the subject programs, work stealing gives a speedup similar to the minimum speedup achieved using static ranges while for the other two thirds, it is about the maximum speedup achieved using static ranges or even more. We believe that the first set

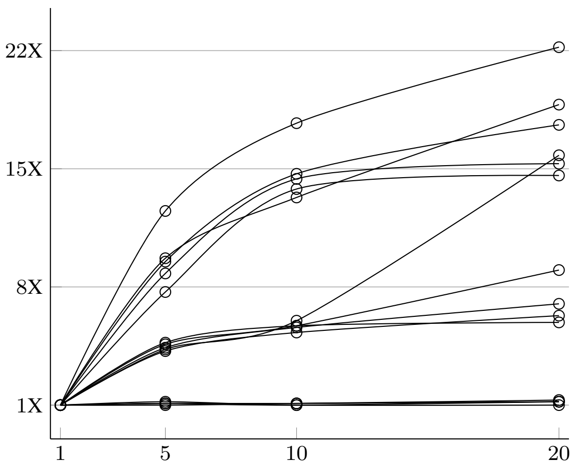


Figure 4. Speedup achieved by 15 programs from GNU Coreutils on different number of workers for ranged symbolic execution with work stealing. These are the worst 5, median 5, and best 5 utilities from Figure 3 based on performance on 10 workers. The worst 5 overlap at or near 1.0X and are hard to distinguish.

of programs have narrow and deep trees while the second set of programs have broad trees that enable better parallelism.

Table 2 shows the results of running work stealing based ranged symbolic execution on a smaller set of 15 programs using 5, 10, and 20 workers with 1 coordinator processor and compares it to the performance of analyzing sequentially. Data for 1 and 10 processors is taken from Table 1. This data is plotted in Figure 4. These are the 5 worst, 5 median, and 5 best performing programs in the first experiment as discussed in Section 4.2. The 5 programs that performed worst in the first experiment do not gain anything from more processors and hardly give any further speedup. Most of the other 10 programs, however, gained more speedup. The speedup possible using any parallel technique for symbolic execution is restricted by the program structure. If a program has a deep and narrow execution tree (e.g., one main path and only branching for error checks), then one or a few paths take nearly as much time as the time for complete analysis. Any scheme that completely checks one path on one processor is unlikely to improve performance of such programs.

4.4 Threats to validity

We tested our technique on one set of programs. It is possible that other programs exhibit different behavior. We mitigate this threat by choosing a suite of medium sized programs and then considering all of them. This can be seen in the results where we achieve a speedup of 1X (no speedup) to over 17X.

We selected random paths as range boundaries. We expect that in real scenarios, it might be more meaningful to divide ranges using tests from some manual test suite. It is possible that such ranges from manual tests provide much worse or much better performance. We mitigate this threat by repeating the random selection multiple times and reporting the range of speedups in both Table 1 and Figure 3.

5. Related Work

Clarke [9] and King [23] pioneered traditional symbolic execution for imperative programs with primitive types. Much progress has been made on symbolic execution during the last decade. PREfix [4] is among the first systems to show the bug finding ability of symbolic execution on real code. Generalized symbolic execution [22] shows how to apply traditional symbolic execution to object-oriented code and uses *lazy initialization* to handle pointer aliasing.

Symbolic execution guided by concrete inputs has been a topic of extensive investigation during the last six years. DART [15] combines concrete and symbolic execution to collect the branch conditions along the execution path. DART negates the last branch condition to construct a new path condition that can drive the function to execute on another path. DART focuses only on path conditions involving integers. To overcome the path explosion in large programs, SMART [14] introduced inter-procedural static analysis techniques to compute procedure summaries and reduce the paths to be explored by DART. CUTE [33] extends DART to handle constraints on references.

EGT [5] and EXE [6] also use the negation of branch predicates and symbolic execution to generate test cases. They increase the precision of symbolic pointer analysis to handle pointer arithmetic and bit-level memory locations. KLEE [7] is the most recent tool from the EGT/EXE family. KLEE is open-sourced and has been used by a variety of users in academia and industry. KLEE works on LLVM byte code [1]. It works on unmodified programs written in C/C++ and has been shown to work for many off the shelf programs. Ranged symbolic execution uses KLEE as an enabling technology.

While approaches such as DART, CUTE, and KLEE are conceptually easy to implement in a parallel setting using forking on every branch, doing so is unlikely to be feasible as it would require spawning processes (or threads with expensive locking) proportional to the number of paths in the program. As observed in previous work [36, 40], the more the number of parallel work items, the poorer the performance because of high overhead. [36] notes the high communication overheads because of exploring paths separately. A scheme like forking for each path would not be efficient because 1) forking across machines is a “very” costly operation, 2) even on the same machine, forking symbolic execution is costly as the address space of the symbolically executed program will incur heavy copy-on-write penalties because of the way symbolic execution explores code paths, and 3) no shared caches for solutions of partial clauses etc. can be made. As an example, KLEE gets more than 10X slower if its caching is disabled.

A couple of recent research projects have proposed techniques for parallel symbolic execution [36, 40]. ParSym [36] parallelized symbolic execution by treating every path exploration as a unit of work and using a central server to distribute work between parallel workers. While this technique implements a direct approach for parallelization [6, 16], it requires communicating symbolic constraints for every branch explored among workers, which incurs a higher overhead. In contrast, static partitioning [40] uses an initial shallow run of symbolic execution to minimize the communication overhead during parallel symbolic execution. The key idea is to create pre-conditions using conjunctions of clauses on path conditions encountered during the shallow run and to restrict symbolic execution by each worker to program paths that satisfy the pre-condition for that worker's path exploration. However, the creation of pre-conditions results in different workers exploring overlapping ranges, which results in wasted effort. Moreover, static partitioning does not use work stealing. In contrast to these existing techniques, ranged symbolic execution uses dynamic load balancing, ensures workers have no overlap (other than on the paths that define range boundaries), and keeps the communication low.

KleeNet [32] uses KLEE to find interaction bugs in distributed applications by running the distributed components under separate KLEE instances and coordinating them using a network model. KleeNet performs separate symbolic execution tasks of each component of the distributed application in parallel. However, it has no mechanism of parallelizing a single symbolic execution task.

Hybrid concolic testing [27] uses random search to periodically guide symbolic execution to increase code coverage. However, it explores overlapping ranges when hopping from symbolic execution in one area of code to another, since no exploration boundaries are defined (other than time out). Ranged symbolic execution can in fact enable a novel form of hybrid concolic testing, which avoids overlapping ranges by hopping outside of the ranges already explored and not re-entering them.

Staged symbolic execution [37] is a technique to apply symbolic execution in stages, rather than the traditional approach of applying it all at once, to compute abstract symbolic inputs that can later be shared across different methods to test them systematically. Staged symbolic execution conceptually divides symbolic execution in horizontal slices called "stages" that can be executed sequentially. On the other hand, ranged symbolic execution conceptually divides symbolic execution in vertical slices called "ranges" that can be explored in parallel.

Directed incremental symbolic execution [30] leverages differences among program versions to optimize symbolic execution of *affected* paths that may exhibit modified behavior. The basic motivation is to avoid symbolically executing paths that have already been explored in a previous program version that was symbolically executed. A reachability analysis is used to identify affected locations, which guide the symbolic exploration. We believe ranged symbolic execution can provide an alternative technique for incremental symbolic execution where program edits are "wrapped" in test pairs that are computed based on the edit locations and the pairs provide the ranges for symbolic execution.

Other tools in the dynamic analysis domain have also seen parallelization efforts. Korat has been parallelized in two ways [28, 35]. Parallel model checkers have also been introduced. Stern and Dill's parallel Murϕ [41] is an example of a parallel model checker. It keeps the set of visited states shared between parallel workers so that the same parts of the state space are not searched by multiple workers. Keeping this set synchronized between the workers results in expensive communication so the algorithm does not scale well.

A similar technique was used by Lerda and Visser [42] to parallelize the Java PathFinder model checker [26]. Parallel version of the SPIN model checker [18] was produced by Lerda and Sisto [25]. More work has been done in load balancing and reducing worker communication in these algorithms [20, 24, 29]. Parallel Randomized State Space Search for JPF by Dwyer et al. [11] takes a different approach with workers exploring randomly different parts of the state space. This often speeds up time to find first error with no worker communication. However when no errors are present, every worker has to explore every state. Parallel search algorithms in general have been studied [17, 19, 21] even earlier.

6. Future work

We envision a number of exciting new research avenues that build on this paper. The notion of sorting execution paths and the corresponding test inputs can enable novel techniques for regression testing, e.g., by using binary search—an elementary algorithm—on a sorted test suite, say to find the *smallest* and *largest* paths that "enclose" the changed code and identify an *impacted* range. Developing the idea of pausing and resuming an analysis using a succinct representation of the analysis state can be generalized to other program analysis techniques to address a key practical problem in program analysis, namely "how to proceed if an analysis run times out?". Ranging the run of an analysis can allow development of novel methods for applying different program analysis techniques in synergy, e.g., where each technique handles its specific range(s), to further scale effective checking of complex programs. The insights into ranged symbolic execution can help develop novel forms of ranged analysis for other techniques, e.g., a run of a software model checker can be ranged [13] using sequences of choices along execution paths, thereby conceptually restricting the run using "vertical" boundaries, which contrasts with the traditional approach of using a "horizontal" boundary, i.e., the search depth bound, and can provide an effective way to deal with the state-space explosion problem.

7. Conclusions

The connection between symbolic execution and test inputs—specifically, to use symbolic execution to generate inputs—was first established over three decades ago, and since then, has undergone extensive research investigation. But this connection is conceptually in just one direction: *from* symbolic execution *to* tests. The key novelty of our work is to establish the connection in the *opposite* direction—from a test input to symbolic execution—specifically, *to use a test input to encode the state of a run of symbolic execution*—and show how this direction enables a number of novel approaches for more effective symbolic execution for test input generation.

The focus of this paper was on our approach to *range* symbolic execution using two tests, which enables (statically and dynamically) partitioning the symbolic execution problem into several sub-problems for scalability. As an enabling technology we leveraged the open-source tool KLEE, which is a state-of-the-art tool for symbolic execution. Experimental results using 71 programs chosen from the widely deployed GNU Coreutils set of Unix utilities show that our approach provides a significant speedup over KLEE. For example, using 10 worker cores, we achieve an average speedup of 6.6X for the 71 programs.

We believe our encoding of the state of an analysis run using a single test input and our ranging of an analysis using two test inputs will provide a foundation for new scalable approaches for more effective symbolic execution. We hope such approaches will also be developed for other analysis techniques, such as software model checking and sound static analysis, and lead to a verification tool-set that enables the development of more reliable software at a much reduced cost.

Acknowledgments

We thank Lingming Zhang and the anonymous reviewers for detailed and helpful comments. This material is based upon work partially supported by the Fulbright program, the National Science Foundation under Grant No. CCF-0845628, and AFOSR grant FA9550-09-1-0351.

References

[1] V. Adve, C. Lattner, M. Brukman, A. Shukla, and B. Gaeke. LLVA: A Low-level Virtual Instruction Set Architecture. In *Proc. 36th International Symposium on Microarchitecture (MICRO)*, pages 205–216, 2003.

[2] S. Anand, C. S. Păsăreanu, and W. Visser. JPF-SE: a Symbolic Execution Extension to Java PathFinder. In *Proc. 13th* , pages 134–138, 2007.

[3] C. Barrett and C. Tinelli. CVC3. In *Proc. 19th International Conference on Computer Aided Verification (CAV)*, pages 298–302, 2007.

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

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

[6] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically Generating Inputs of Death. In *Proc. 13th Conference on Computer and Communications Security (CCS)*, pages 322–335, 2006.

[7] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. 8th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–224, 2008.

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

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

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

[11] M. B. Dwyer, S. Elbaum, S. Person, and R. Purandare. Parallel Randomized State-Space Search. In *Proc. 2007 International Conference on Software Engineering (ICSE)*, pages 3–12, 2007.

[12] B. Elkarablieh, I. Garcia, Y. L. Suen, and S. Khurshid. Assertion-based Repair of Complex Data Structures. In *Proc. 22nd International Conference on Automated Software Engineering (ASE)*, pages 64–73, 2007.

[13] D. Funes, J. H. Siddiqui, and S. Khurshid. Ranged model checking. Under submission.

[14] P. Godefroid. Compositional Dynamic Test Generation. In *Proc. Symposium on Principles of Programming Languages (POPL)*, pages 47–54, 2007.

[15] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proc. 2005 Conference on Programming Languages Design and Implementation (PLDI)*, pages 213–223, 2005.

[16] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated Whitebox Fuzz Testing. In *Proc. Network and Distributed System Security Symposium (NDSS)*, 2008.

[17] A. Grama and V. Kumar. State of the Art in Parallel Search Techniques for Discrete Optimization Problems. *IEEE Transactions on Knowledge and Data Engineering* , 11(1):28–35, Jan. 1999.

[18] G. J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering* , 23(5):279–295, May 1997.

[19] V. K. Janakiram, D. P. Agrawal, and R. Mehrotra. A Randomized Parallel Backtracking Algorithm. *IEEE Transactions on Computers*, 37(12):1665–1676, Dec. 1988.

[20] M. D. Jones and J. Sorber. Parallel Search for LTL Violations. *International Journal Software Tools Technology Transfer* , 7 (1):31–42, Feb. 2005.

[21] R. M. Karp and Y. Zhang. Randomized Parallel Algorithms for Backtrack Search and Branch-and-bound Computation. *Journal of the ACM*, 40(3):765–789, July 1993.

[22] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized Symbolic Execution for Model Checking and Testing. In *Proc. 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 553–568, 2003.

[23] J. C. King. Symbolic Execution and Program Testing. *Communications ACM*, 19(7):385–394, July 1976.

[24] R. Kumar and E. G. Mercer. Load Balancing Parallel Explicit State Model Checking. *Electronics Notes Theory Computer Science* , 128(3):19–34, Apr. 2005.

[25] F. Lerda and R. Sisto. Distributed-Memory Model Checking with SPIN. In *Proc. 5th International SPIN Workshop on Model Checking of Software* , pages 22–39, 1999.

[26] F. Lerda and W. Visser. Addressing Dynamic Issues of Program Model Checking. In *Proc. 8th International SPIN Workshop on Model Checking of Software* , pages 80–102, 2001.

[27] R. Majumdar and K. Sen. Hybrid Concolic Testing. In *Proc. 29th International Conference on Software Engineering (ICSE)*, pages 416–426, 2007.

[28] S. Misailovic, A. Milicevic, N. Petrovic, S. Khurshid, and D. Marinov. Parallel Test Generation and Execution with Korat. In *Proc. 6th joint meeting of the European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 135–144, 2007.

[29] R. Palmer and G. Gopalakrishnan. A Distributed Partial Order Reduction Algorithm. In *Proc. 22nd International Conference on Formal Techniques for Networked and Distributed Systems (FORTE)*, page 370, 2002.

[30] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed Incremental Symbolic Execution. In *Proc. 2011 Conference on Programming Languages Design and Implementation (PLDI)*, pages 504–515, 2011.

[31] D. A. Ramos and D. R. Engler. Practical, Low-Effort Equivalence Verification of Real Code. In *Proc. 23rd International Conference on Computer Aided Verification (CAV)*, pages 669–685, 2011.

[32] R. Sasnauskas, O. Landsiedel, M. H. Alizai, C. Weise, S. Kowalewski, and K. Wehrle. KleeNet: Discovering Insidious Interaction Bugs in Wireless Sensor Networks Before Deployment. In *Proc. 9th International Conference on Information Processing in Sensor Networks (ISPN 2010)*, pages 186–196, 2010.

[33] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proc. 5th joint meeting of the European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 263–272, 2005.

[34] C. Seo, S. Malek, and N. Medvidovic. Component-Level Energy Consumption Estimation for Distributed Java-Based Software Systems. In *Proc. 11th International Symposium on Component-Based Software Engineering*, pages 97–113, 2008.

[35] J. H. Siddiqui and S. Khurshid. PKorat: Parallel Generation of Structurally Complex Test Inputs. In *Proc. 2nd International Conference on Software Testing Verification and Validation (ICST)*, pages 250–259, 2009.

[36] J. H. Siddiqui and S. Khurshid. ParSym: Parallel Symbolic Execution. In *Proc. 2nd International Conference on Software Technology and Engineering (ICSTE)*, pages V1: 405–409, 2010.

[37] J. H. Siddiqui and S. Khurshid. Staged Symbolic Execution. In *Proc. Symposium on Applied Computing (SAC): Software Verification and Testing Track (SVT)*, 2012.

[38] M. Snir and S. Otto. *MPI-The Complete Reference: The MPI Core*. MIT Press, 1998.

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

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

[41] U. Stern and D. L. Dill. Parallelizing the Murphi Verifier. In *Proc. 9th International Conference on Computer Aided Verification*, pages 256–278, 1997.

[42] W. Visser, K. Havelund, G. Brat, S. P. Park, and F. Lerda. Model Checking Programs. *Automated Software Engineering Journal* , 10(2):203–232, Apr. 2003.