Directed symbolic execution

Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks

Computer Science Department, University of Maryland, College Park
{kkma,khooyp,jfoster,mwh}@cs.umd.edu

Abstract. In this paper, we study the problem of automatically finding program executions that reach a particular target line. This problem arises in many debugging scenarios; for example, a developer may want to confirm that a bug reported by a static analysis tool on a particular line is a true positive. We propose two new *directed* symbolic execution strategies that aim to solve this problem: *shortest-distance symbolic execution (SDSE)* uses a distance metric in an interprocedural control flow graph to guide symbolic execution toward a particular target; and *call-chain-backward symbolic execution (CCBSE)* iteratively runs forward symbolic execution, starting in the function containing the target line, and then jumping backward up the call chain until it finds a feasible path from the start of the program. We also propose a hybrid strategy, Mix-CCBSE, which alternates CCBSE with another (forward) search strategy. We compare these three with several existing strategies from the literature on a suite of six GNU coreutils programs. We find that SDSE performs extremely well in many cases but may fail badly. CCBSE also performs quite well, but imposes additional overhead that sometimes makes it slower than SDSE. Considering all our benchmarks together, Mix-CCBSE performed best on average, combining to good effect the features of its constituent components.

1 Introduction

In this paper, we study the *line reachability problem*: given a target line in the program, can we find a realizable path to that line? Since program lines can be guarded by conditionals that check arbitrary properties of the current program state, this problem is equivalent to the very general problem of finding a path that causes the program to enter a particular state [12]. The line reachability problem arises naturally in several scenarios. For example, users of static-analysis-based bug finding tools need to *triage* the tools' bug reports—determine whether they correspond to actual errors—and this task often involves checking line reachability. As another example, a developer might receive a report of an error at some particular line (e.g., an assertion failure that resulted in an error message at that line) without an accompanying test case. To reproduce the error, the developer needs to find a realizable path to the appropriate line. Finally, when trying to understand an unfamiliar code base, it is often useful to discover under what circumstances particular lines of code are executed.

Symbolic execution is an attractive approach to solving line reachability: by design, symbolic executors are *complete*, meaning any path they find is realizable. Symbolic executors work by running the program, computing over both concrete values and expressions that include *symbolic values*, which are unknowns that range over various sets of values, e.g., integers, strings, etc. [17, 2, 15, 29]. When a symbolic executor encounters a conditional whose guard depends on a symbolic value, it invokes a theorem prover (our implementation uses the SMT solver STP [10]) to determine which branches are feasible. If both are, the symbolic execution conceptually forks, exploring both branches.

However, symbolic executors cannot explore all program paths, and hence must make heuristic choices to prioritize path exploration. Our work focuses on finding paths that reach certain lines in particular, whereas most prior work has focused on finding paths to increase code coverage [11, 5, 4, 24, 3, 34]. We are aware of one previously proposed approach, *execution synthesis* (ESD) [36], for using symbolic execution to solve the line reachability problem; we compare ESD to our work in Section 3.

We propose two new *directed* symbolic execution search strategies for line reachability. First, we propose *shortest-distance symbolic execution* (SDSE), which prioritizes the path with the shortest distance to the target line as computed over an interprocedural control-flow graph (ICFG). Variations of this heuristic can be found in existing symbolic executors—in fact, SDSE is inspired by the heuristic used in the coverage-based search strategy from KLEE [4]—but, as far as we are aware, the strategy we present has not been specifically described nor has it been applied to directed symbolic execution. In Section 2.2 we describe how distance can be computed context-sensitively using *PN* grammars [32, 9, 30].

Second, we propose *call-chain-backward symbolic execution (CCBSE)*, which starts at the target line and works backward until it finds a realizable path from the start of the program, using standard forward (interprocedural) symbolic execution as a subroutine. More specifically, suppose the target line ℓ is inside function f. CCBSE begins forward symbolic execution from the start of f, yielding a set of partial interprocedural paths $\overline{p}_f$ that start at f, possibly call other functions, and lead to ℓ; in a sense, these partial paths summarize selected behavior of f. Next, CCBSE runs forward symbolic execution from the start of each function g that calls f, searching for paths that end at calls to f. For each such path p, it attempts to continue down paths p' in $\overline{p}_f$ until reaching ℓ, adding all feasible extended paths $p + p'$ to $\overline{p}_g$. The process continues backward up the call chain until CCBSE finds a path from the start of the program to ℓ. Notice that by using partial paths to summarize function behavior, CCBSE can reuse the machinery of symbolic execution to concatenate paths together. This is technically far simpler than more standard approaches that use some formal language to explicitly summarize function behavior in terms of parameters, return value, global variables, and the heap (including pointers and aliasing).

The key insight motivating CCBSE is that the closer forward symbolic execution starts relative to the target line, the better the chance it finds paths to that line. If we are searching for a line that is only reachable on a few paths

along which many branches are possible, then combinatorially there is a very small chance that a standard symbolic executor will make the right choices and find that line. By starting closer to the line we are searching for, CCBSE explores shorter paths with fewer branches, and so is more likely to reach that line.

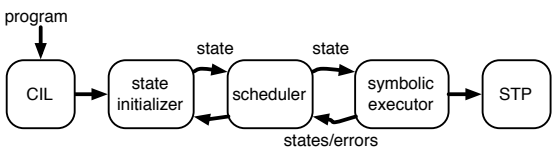
CCBSE imposes some additional overhead, and so it does not always perform as well as a forward execution strategy. Thus, we also introduce *mixed-strategy CCBSE (Mix-CCBSE)*, which combines CCBSE with another forward search. In Mix-CCBSE, we alternate CCBSE with some forward search strategy S. If S encounters a path p that was constructed in CCBSE, we try to follow p to see if we can reach the target line, in addition to continuing S normally. In this way, Mix-CCBSE can perform better than CCBSE and S run separately—compared to CCBSE, it can jump over many function calls from the program start to reach the paths being constructed; and compared to S, it can short-circuit the search once it encounters a path built up by CCBSE.

We implemented SDSE, CCBSE, and Mix-CCBSE in Otter, a C source code symbolic executor we previously developed [31]. We also extended Otter with two popular forward search strategies from KLEE [4] and SAGE [13], and for a baseline, we implemented a random path search that flips a coin at each branch. We evaluated the effectiveness of our directed search strategies on the line reachability problem, comparing against the existing search strategies. We ran each strategy on 6 different GNU Coreutils programs [6], looking in each program for one line that contains a previously identified fault. We also compared the strategies on synthetic examples intended to illustrate the strengths of SDSE and CCBSE. We found that SDSE performs extremely well on many programs, but it can fail completely under certain program patterns. CCBSE has performance comparable to standard search strategies but is often somewhat slower due to the overhead of checking path feasibility. Mix-CCBSE performs best of all across all benchmarks, particularly when using KLEE as its forward search strategy, since it exploits the best features of CCBSE and forward search. These results suggest that directed symbolic execution is a practical and effective approach to solving the line reachability problem.

2 Directed Symbolic Execution

In this section we present SDSE, CCBSE, and Mix-CCBSE. We will explain them in terms of their implementation in Otter, our symbolic execution framework, to make our explanations concrete (and to save space), but the ideas apply to any symbolic execution tool [17, 11, 4, 16].

Figure 1 diagrams the architecture of Otter and gives pseudocode for its main scheduling loop. Otter uses CIL [27] to produce a control-flow graph from the input C program. Then it calls a *state initializer* to construct an initial symbolic execution *state*, which it stores in worklist, used by the scheduler. A state includes the stack, heap, program counter, and path taken to reach the current position. In traditional symbolic execution, which we call *forward* symbolic execution, the initial state begins execution at the start of main. The scheduler extracts a




```
1  scheduler()
2    while (worklist nonempty)
3      s_0 = pick(worklist)
4      for s ∈ step(s_0) do
5        if (s is incomplete)
6          put(worklist,s)
7        manage_targets(s)
```

Fig. 1. The architecture of the Otter symbolic execution engine.

state from the worklist via pick and symbolically executes the next instruction by calling step. As Otter executes instructions, it may encounter conditionals whose guards depend on symbolic values. At these points, Otter queries STP [10], an SMT solver, to see if legal, concrete representations of the symbolic values could make either or both branches possible, and whether an error such as an assertion failure may occur. The symbolic executor will return these states to the scheduler, and those that are *incomplete* (i.e., non-terminal) are added back to the worklist. The call to manage_targets is just for guiding CCBSE's backward search (it is a no-op for other strategies), and is discussed further below.

2.1 Forward symbolic execution

Different forward symbolic execution strategies are distinguished by their implementation of the pick function. In Otter we have implemented, among others, three search strategies described in the literature:

Random Path (RP) [3, 4] is a probabilistic version of breadth-first search. RP randomly chooses from the worklist states, weighing a state with a path of length n by 2^{-n}. Thus, this approach favors shorter paths, but treats all paths of the same length equally.

KLEE [4] uses a round-robin of RP and what we call *closest-to-uncovered*, which computes the distance between the end of each state's path and the closest uncovered node in the interprocedural control-flow graph and then randomly chooses from the set of states weighed inversely by distance. To our knowledge, KLEE's algorithm has not been described in detail in the literature; we studied it by examining KLEE's source code [18].

SAGE [13] uses a coverage-guided *generational* search to explore states in the execution tree. At first, SAGE runs the initial state until the program terminates by randomly choosing a state to run whenever the symbolic execution core returns multiple states. It stores the remaining states into the worklist as the *first generation children*. Next, SAGE runs each of the first generation children to completion, in the same manner as the initial state, but separately grouping the grandchildren by their first generation parent. After exploring the first generation, SAGE explores subsequent generations (children of the first generation, grandchildren of the first generation, etc.) in a more intermixed fashion, using a block coverage heuristic to determine which generations to explore first.

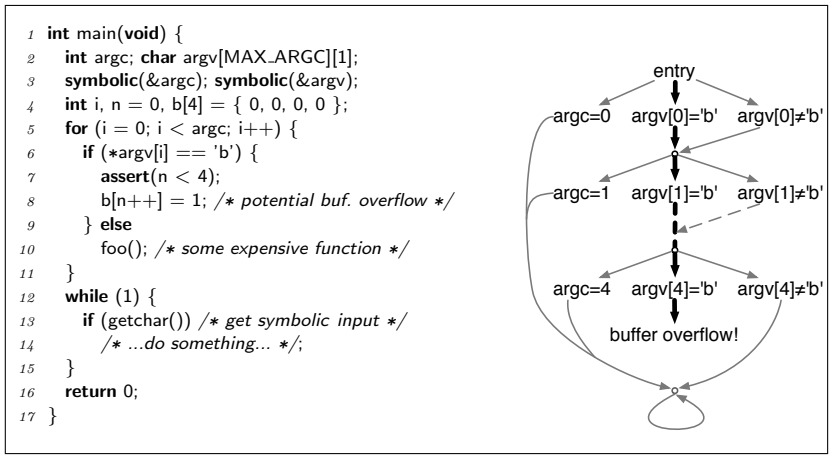


Fig. 2. Example illustrating SDSE's potential benefit.

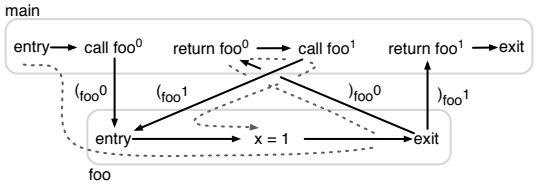
2.2 Shortest-distance symbolic execution

The basic idea of SDSE is to prioritize program branches that correspond to the shortest path-to-target in the ICFG. To illustrate how SDSE works, consider the code in Figure 2, which performs command-line argument processing followed by some program logic, a pattern common to many programs. This program first enters a loop that iterates up to argc times, processing the i^{th} command-line argument in argv during iteration i. If the argument is 'b', the program sets b[n] to 1 and increments n (line 8); otherwise, the program calls foo. A potential buffer overflow could occur at line 8 when more than four arguments are 'b'; we add an assertion on line 7 to identify when this overflow would occur. After the arguments are processed, the program enters a loop that reads and processes character inputs (lines 12 onward).

Suppose we would like to reason about a possible failure of the assertion. Then we can run this program with symbolic inputs, which we identify with the calls on line 3 to the special built-in function symbolic. The right half of the figure illustrates the possible program paths the symbolic executor can explore on the first five iterations of the argument-processing loop. Notice that for five loop iterations there is only one path that reaches the failing assertion out of $\sum_{n=0}^{4} 3 \times 2^n = 93$ total paths. Moreover, the assertion is not reachable once exploration has advanced past the argument-processing loop.

In this example, RP would have only a small chance of finding the overflow, spending most of its time exploring paths shorter than the one that leads to the buffer overflow. A symbolic executor using KLEE or SAGE would focus on increasing coverage to all lines, wasting significant time exploring paths through the loop at the end of the program, which does not influence this buffer overflow.

In contrast, SDSE works very well in this example, with line 7 set as the target. Consider the first iteration of the loop. The symbolic executor will branch upon reaching the loop guard, and will choose to execute the first instruction of the loop, which is two lines away from the assertion, rather than the first instruc-



(a) Example PN-path in an interprocedural CFG.

$$
\begin{array}{rcl}
PN & \rightarrow & P\ N \\
P & \rightarrow & S\ P \\
 & | &)_i\ P \\
 & | & \epsilon
\end{array}
\qquad
\begin{array}{rcl}
N & \rightarrow & S\ N \\
 & | & (_i\ N \\
 & | & \epsilon \\
S & \rightarrow & (_i\ S\ )_i \\
 & | & S\ S \\
 & | & \epsilon
\end{array}
$$

(b) Grammar of PN paths.

Fig. 3. SDSE distance computation.

tion after the loop, which can no longer reach the assertion. Next, on line 6, the symbolic executor takes the true branch, since that reaches the assertion itself immediately. Then, determining that the assertion is true, it will run the next line, since it is only three lines away from the assertion and hence closer than paths that go through foo (which were deferred by the choice to go to the assertion). Then the symbolic executor will return to the loop entry, repeating the same process for subsequent iterations. As a result, SDSE explores the central path shown in bold in the figure, and thereby quickly find the assertion failure.

Implementation. SDSE is implemented as a pick function from Figure 1. As mentioned, SDSE chooses the state on the worklist with the shortest *distance to target*. Within a function, the distance is just the number of edges between statements in the control flow graph (CFG). To measure distances across function calls we count edges in an interprocedural control-flow graph (ICFG) [21], in which function call sites are split into *call nodes* and *return nodes*, with *call edges* connecting call nodes to function entries and *return edges* connecting function exits to return nodes. For each call site i, we label call and return edges by $(_i$ and $)_i$, respectively. Figure 3(a) shows an example ICFG for a program in which main calls foo twice; here call i to foo is labeled foo^i.

We define the distance-to-target metric to be the length of the shortest path in the ICFG from an instruction to the target, such that the path contains no mismatched calls and returns. Formally, we can define such paths as those whose sequence of edge labels form a string produced from the PN nonterminal in the grammar shown in Figure 3(b). In this grammar, developed by Reps [32] and later named by Fähndrich et al [9, 30], S-paths correspond to those that exactly match calls and returns; N-paths correspond to entering functions only; and P-paths correspond to exiting functions only. For example, the dotted path in Figure 3(a) is a PN-path: it traverses the matching $(_{foo^0}$ and $)_{foo^0}$ edges, and then traverses $(_{foo^1}$ to the target. Notice that we avoid conflating edges of different call sites by matching $(_i$ and $)_i$ edges, and thus we can statically compute a context-sensitive distance-to-target metric.

PN-reachability was previously used for conservative static analysis [9, 30, 19]. However, in SDSE, we are always asking about PN-reachability from the current instruction. Hence, rather than solve reachability for an arbitrary initial P-path segment (which would correspond to asking about distances from the current instruction in all calling contexts of that instruction), we restrict the ini-

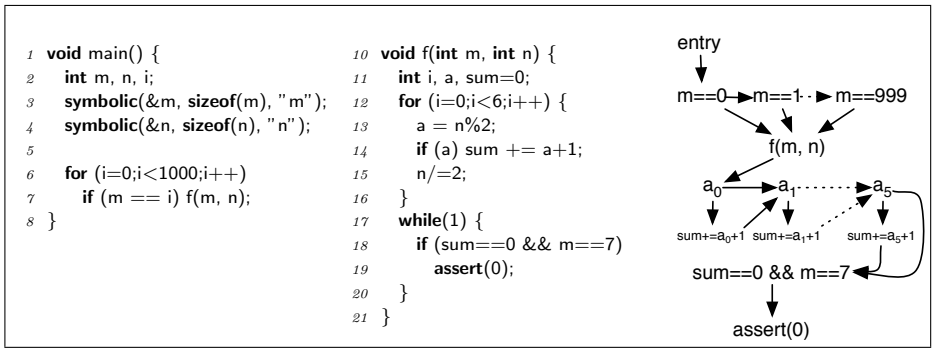


Fig. 4. Example illustrating CCBSE's potential benefit.

tial P-path segment to the functions on the current call stack. For performance, we statically pre-compute N-path and S-path distances for all instructions to the target and combine them with P-path distances on demand.

2.3 Call-chain-backward symbolic execution

SDSE is often very effective, but there are cases on which it does not do well—in particular, SDSE is less effective when there are many potential paths to the target line, but there are only a few, long paths that are realizable. In these situations, CCBSE can sometimes work dramatically better.

To see why, consider the code in Figure 4. This program initializes `m` and `n` to be symbolic and then loops, calling `f(m, n)` when `m == i` for `i` $\in [0, 1000)$. For non-negative values of `n`, the loop in lines 12–16 iterates through `n`'s least significant bits (stored in `a` during iteration), incrementing `sum` by `a+1` for each non-zero `a`. Finally, if `sum == 0` and `m == 7`, the failing assertion on line 19 is reached. Otherwise, the program falls into an infinite loop, as `sum` and `m` are never updated in the loop.

RP, KLEE, SAGE, and SDSE all perform poorly on this example. SDSE gets stuck at the very beginning: in `main`'s for-loop, it immediately steps into `f` when `m == 0`, as this is the "fastest" way to reach the assertion inside `f` according to the ICFG. Unfortunately, the guard of the assertion is never satisfied when `m` is 0, and therefore SDSE gets stuck in the infinite loop. SAGE is very likely to get stuck, because the chance of SAGE's first generation entering `f` with the right argument (`m == 7`) is extremely low, and SAGE always runs its first generation to completion, and hence will execute the infinite loop forever. RP and KLEE will also reach the assertion very slowly, since they waste time executing `f` where `m`$\neq 7$; none of these paths lead to the assertion failure.

In contrast, CCBSE begins by running `f` with both parameters `m` and `n` set to symbolic, as CCBSE does not know what values might be passed to `f`. Hence, CCBSE will potentially explore all 2^6 paths induced by the for loop, and one of them, say p, will reach the assertion. When p is found, CCBSE will jump to `main` and explore various paths that reach the call to `f`. At the call to `f`, CCBSE will

```
8  manage_targets (s)
9    (sf,p) = path(s)
10   if pc(p) ∈ targets
11     update_paths(sf, p)
12   else if pc(p) = callto(f) and has_paths(f)
13     for p′ ∈ get_paths(f)
14       if (p + p′ feasible)
15         update_paths(sf, p + p′)
16 update_paths (sf, p)
17   if not(has_paths(sf))
18     add_callers(sf,worklist)
19   add_path(sf, p);
```

Fig. 5. Target management for CCBSE.

follow p to short-circuit the evaluation through f (in particular, the 2^6 branches induced by the for-loop), and thus quickly find a realizable path to the failure.

Implementation. CCBSE is implemented in the manage_targets and pick functions from Figure 1. Otter states s, returned by pick, include the function f in which symbolic execution started, which we call the *origin function*. Thus, traditional symbolic execution states always have main as their origin function, while CCBSE allows different origin functions. In particular, CCBSE begins by initializing states for functions containing target lines.

To start symbolic execution at an arbitrary function Otter must initialize symbolic values for the function's inputs (parameters and global variables). Integer-valued inputs are initialized to symbolic words, and pointers are represented using *conditional pointers*, manipulated using Morris's general axiom of assignment [1, 26]. To support recursive data structures, Otter initializes pointers lazily—we do not actually create conditional pointers until a pointer is used, and we only initialize as much of the memory map as is required. When initialized, pointers are set up as follows: for inputs p of type *pointer to type* T, we construct a conditional pointer such that p may be null or p may point to a fresh symbolic value of type T. If T is a primitive type, we also add a disjunct in which p may point to any element of an array of 4 fresh values of type T. This last case models parameters that are pointers to arrays, and we restrict its use to primitive types for performance reasons. In our experiments, we have not found this restriction to be problematic. This strategy for initializing pointers is unsound in that CCBSE could miss some targets, but final paths CCBSE produces are always feasible since they ultimately connect back to main.

The pick function works in two steps. First, it selects the origin function to execute and then it selects a state with that origin. For the former, it picks the function f with the shortest-length call chain from main. For non-CCBSE the origin will always be main. At the start of CCBSE with a single target, the origin will be the one containing the target; as execution continues there will be more choices—picking the "shortest to main" ensures that we move backward from target functions toward main. After selecting the origin function f, pick chooses one of f's states according to some forward search strategy. We write CCBSE(S) to denote CCBSE using forward search strategy S.

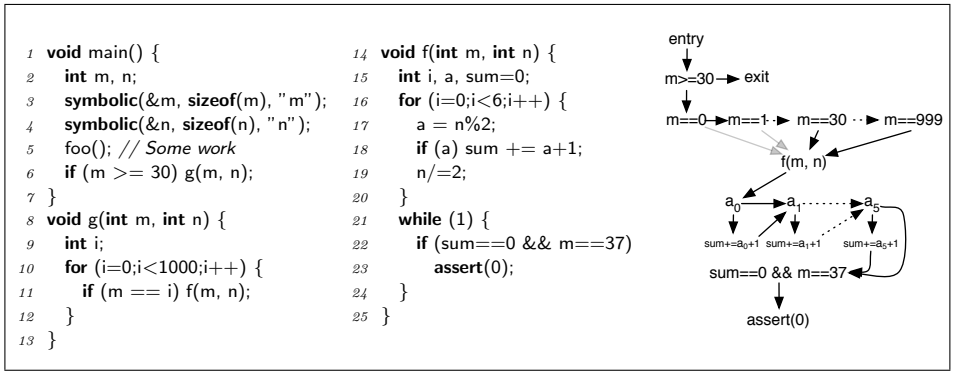


Fig. 6. Example illustrating Mix-CCBSE's potential benefit.

The manage_targets(s) function is given in Figure 5. Recall from Figure 1 that s has already been added to the worklist for additional, standard forward search; the job of manage_targets is to record which paths reach the target line and to try to connect s with path suffixes previously found to reach the target. The manage_targets function extracts from s both the origin function sf and the (interprocedural) *path* p that has been explored from sf to the current point. This path contains all the decisions made by the symbolic executor at condition points. If path p's end (denoted pc(p)) has reached a target (line 10), we associate p with sf by calling update_paths; for the moment one can think of this function as adding p to a list of paths that start at sf and reach targets. Otherwise, if the path's end is at a call to some function f, and f itself has paths to targets, then we may possibly extend p with one or more of those paths. So we retrieve f's paths, and for each one p' we see whether concatenating p to p' (written $p + p'$) produces a feasible path. If so, we add it to sf's paths. Feasibility is checked by attempting to symbolically execute p' starting in p's state s.

Now we turn to the implementation of update_paths. This function simply adds p to sf's paths (line 19), and if sf did not previously have any paths, it will create initial states for each of sf's callers (pre-computed from the call graph) and add these to the worklist (line 17). Because these callers will be closer to main, they will be subsequently favored by pick when it chooses states.

2.4 Mixing CCBSE with forward search

While CCBSE may find a path more quickly, it comes with a cost: its queries tend to be more complex than in forward search, and it can spend significant time trying paths that start in the middle of the program but are ultimately infeasible. Consider Figure 6, a modified version of the code in Figure 4. Here, main calls function g, which acts as main did in Figure 4, with some m >= 30 (line 6), and the assertion in f is reachable only when m == 37 (line 22). All other strategies fail in the same manner as they do in Figure 4.

However, CCBSE also fails to perform well here, as it does not realize that m is at least 30, and therefore considers ultimately infeasible conditions $0 \leq$ m $\leq$

36 in f. With Mix-CCBSE, however, we conceptually start forward symbolic execution from main at the same time that CCBSE ("backward search") is run. As before, the backward search will gets stuck in finding a path from g's entry to the assertion. However, in the forward search, g is called with m $\geq$ 30, and therefore f is always called with m $\geq$ 30, making it hit the right condition m == 37 very soon thereafter. Notice that, in this example, the backward search must find the path from f's entry to the assertion *before* f is called with m == 37 in the forward search in order for the two searches to match up (e.g., there are enough instructions to run in line 5). Should this not happen, Mix-CCBSE degenerates to its constituents running independently in parallel, which is the worst case.

Implementation. We implement Mix-CCBSE with a slight alteration to pick. At each step, we decide whether to use regular forward search or CCBSE next, splitting the strategies 50/50 by time spent. We compute time heuristically as $50 \times$ (no. of solver calls) + (no. of instructions executed), taking into account the higher cost of solver queries over instruction executions.[1]

3 Experiments

We evaluated our directed search strategies by comparing their performance on the small example programs from Section 2 and on bugs reported in six programs from GNU Coreutils version 6.10. These bugs were previously discovered by KLEE [4]. All experiments were run on a machine with six 2.4Ghz quad-core Xeon E7450 processors and 48GB of memory, running 64-bit Linux 2.6.26. We ran 16 tests in parallel, observing minimal resource contention. The tests required less than 2 days of elapsed time. Total memory usage was below 1GB per test.

The results are presented in Table 1. Part (a) of the table gives the results for our directed search strategies. For comparison, we also implemented an intraprocedural variant of SDSE that ignores call-chains: if the target is not in the current function, then the distance-to-target is ∞. We refer to the intraprocedural variant as IntraSDSE, and to standard SDSE as InterSDSE. This table lists three variants of CCBSE, using RP, InterSDSE, or IntraSDSE as the forward strategy. In the last two cases, we modified Inter- and IntraSDSE slightly to compute shortest distances to the target line *or* to the functions reached in CCBSE's backward search. This allows those strategies to take better advantage of CCBSE (otherwise they would ignore CCBSE's search in determining which paths to take).

Part (b) of the table gives the results from running KLEE version r130848 [18], and part (c) gives the results for forward search strategies implemented in Otter, both by themselves and mixed with CCBSE(RP). We chose CCBSE(RP) because it was the best overall of the three from part (a), and because RP is the fastest of the forward-only strategies in part (c). We write Mix-CCBSE(S) to denote the mixed strategy where S is the forward search strategy and CCBSE(RP)

[1] We could also use wall-clock time, however, this leads to non-deterministic, non-reproducible results. We opted to use our heuristic for reproducibility.

	Inter-SDSE	Intra-SDSE	CCBSE(X) where X is			KLEE
			RP	InterSDSE	IntraSDSE	
Figure 2	0.4 0.0	0.4 0.0(5)	16.2 2.4(6)	0.5 0.0(1)	0.4 0.0(3)	2.6 0.0(7)
Figure 4	∞	∞	60.8 7.8(4)	7.3 1.2(3)	7.2 1.0(4)	∞
Figure 6	∞	∞	∞	∞	∞	∞
mkdir	34.7 19.7(10)	∞	163.0 42.5	150.3 93.4	150.7 93.9	∞
mkfifo	13.1 0.4	∞	70.2 17.3	49.7 21.8	49.3 23.2(1)	274.2 315.6(9)
mknod	∞	∞	216.5 60.7	∞	∞	851.6 554.2(8)
paste	12.6 0.5	56.4 5.4	26.0 0.5(1)	31.0 4.8	32.1 4.0	30.6 9.7(8)
ptx	18.4 0.6(4)	103.5 19.7(1)	24.2 0.7(1)	24.5 0.9(3)	24.1 1.1(2)	93.8 81.7(7)
seq	12.1 0.4(1)	∞	30.9 1.4	369.3 425.9(6)	391.8 411.1(6)	38.2 14.5(8)
Total	1891.0	7360.0	530.9	2424.8	2448.0	3088.55

(a) Directed search strategies (b) KLEE

	Otter-KLEE		Otter-SAGE		Random Path	
	Pure	w/CCBSE	Pure	w/CCBSE	Pure	w/CCBSE
Figure 2	101.1 57.5(4)	104.8 57.3(5)	∞	∞	15.3 2.2(6)	16.1 2.6(6)
Figure 4	579.7 ∞	205.5 133.1(9)	∞	∞	160.1 6.4(11)	80.6 177.2(9)
Figure 6	587.8 ∞	147.6 62.6(7)	∞	∞	169.8 9.1(8)	106.8 11.2(4)
mkdir	168.9 31.0	124.7 12.1(2)	365.3 354.2(5)	1667.7 ∞	143.5 5.3	136.4 7.9
mkfifo	41.7 5.2(1)	38.2 4.6	77.6 101.1(2)	251.9 257.0(8)	59.4 3.7	52.7 1.8(1)
mknod	174.8 24.1	93.1 12.7	108.5 158.7(5)	236.4 215.0(5)	196.7 3.9(2)	148.9 11.8
paste	22.6 0.5(4)	28.6 0.9(3)	54.9 36.2(5)	60.4 52.1(3)	22.1 0.6	27.3 1.0(1)
ptx	33.2 3.9	27.1 2.7	∞	∞	28.9 0.8	28.1 1.1(2)
seq	354.8 94.3(1)	49.3 5.1(1)	∞	288.8 ∞	170.8 3.7(3)	35.9 1.4(1)
Total	795.8	360.9	4206.4	4305.3	621.3	429.4

(c) Undirected search strategies and their mixes with CCBSE(RP)

Table 1. Statistics from benchmark runs. For each coreutils program and for the total, the fastest two times are highlighted. Key: Median SIQR(Outliers) ∞ : time out

is the backward strategy. We did not directly compare against execution synthesis (ESD) [36], a previously proposed directed search strategy; at the end of this section we relate our results to those reported in the ESD paper.

We found that the randomness inherent in most search strategies and in the STP theorem prover introduces tremendous variability in the results. Thus, we ran each strategy/target condition 41 times, using integers 1 to 41 as random seeds for Otter. (We were unable to find a similar option in KLEE, and so simply ran it 41 times.) The main numbers in Table 1 are the medians of these runs, and the small numbers are the semi-interquartile range (SIQR). The number outliers—which fall 3×SIQR below the lower quartile or above the upper quartile, if non-zero—is given in parentheses. We ran each test for at most 600 seconds for the synthetic examples, and at most 1,800 seconds for the Coreutils programs. The median is ∞ if more than half the runs timed out, while the

SIQR is ∞ if more than one quarter of the runs timed out. We highlight the fastest two times in each row.

3.1 Synthetic programs

The first three rows in Table 1 give the results from the examples in Figures 2, 4, and 6. In all cases the programs behaved as predicted.

For the program in Figure 2, both InterSDSE and IntraSDSE performed very well. Since the target line is in main, CCBSE(*SDSE) is equivalent to *SDSE, so those variants performed equally well. Otter-KLEE took much longer to find the target, with more than a quarter of the runs timing out, whereas Otter-SAGE timed out for more than half the runs. RP was able to find the target, but it took much longer than *SDSE. Note that CCBSE(RP) degenerates to RP in this example, and runs in about the same time as RP. Lastly, KLEE performed very well also, although it was still slower than *SDSE in this example.

For the program in Figure 4, CCBSE(InterSDSE) and CCBSE(IntraSDSE) found the target line quickly, while CCBSE(RP) did so in reasonable amount of time. CCBSE(*SDSE) were much more efficient, because with these strategies, after each failing verification of f(m,n) (when $0 \leq \mathsf{m} < 7$), the *SDSE strategies chose to try f(m+1,n) rather than stepping into f, as f is a target added by CCBSE and is closer from any point in main than the assertion in f is.

For the program in Figure 6, Mix-CCBSE(RP) and Mix-CCBSE(Otter-KLEE) performed the best among all strategies, as expected. However, Mix-CCBSE(Otter-SAGE) performed far worse. This is because its forward search (Otter-SAGE) got stuck in one value of m in the very beginning, and therefore it and the backward search did not match up.

3.2 GNU Coreutils

The lower rows of Table 1 give the results from the Coreutils programs. The six programs we analyzed contain a total of 2.4 kloc and share a common library of about 30 kloc. For each bug, we manually added a corresponding failing assertion to the program, and set that as the target line. For example, the Coreutils program seq has a buffer overflow in which an index i accesses outside the bounds of a string fmt [25]. Thus, just before this array access, we added an assertion **assert**(i<strlen(fmt)) to indicate the overflow. Each assertion has a call-chain distance from main ranging from two to seven. Note that Otter does have built-in detection of buffer overflows and similar errors, but for our experiments we do not use this feature to identify valid targets for line reachability.

The Coreutils programs receive input from the command line and from standard input. We initialized the command line as in KLEE [4]: given a sequence of integers $n_1, n_2, \cdots, n_k$, Otter sets the program to have (excluding the program name) at least 0 and at most k arguments, where the ith argument is a symbolic string of length n_i. All of the programs we analyzed used $(10, 2, 2)$ as the input sequence, except mknod, which used $(10, 2, 2, 2)$. Standard input is modeled as an unbounded stream of symbolic values.

```
1 int main(int argc, char** argv) {
2     while ((optc = getopt_long (argc, argv, opts, longopts, NULL)) != −1) { ... } ...
3     if (/* some condition */) quote(...);
4     ...
5     if (/* another condition */) quote(...);
6 }
```

Fig. 7. Code pattern in mkdir, mkfifo and mknod

Coreutils programs make extensive use of the C standard library. To support them, we implemented a partial model of POSIX system calls on top of an in-memory file system, and combined this with the newlib C standard library implementation [28]. All this code is written in C, so Otter executes it as it would any other source code.

Analysis. We can see clearly from the shaded boxes in Table 1 that InterSDSE performed extremely well, achieving the fastest running times on five of the six programs. However, InterSDSE timed out on mknod. Examining this program, we found it shares a similar structure with mkdir and mkfifo, sketched in Figure 7. These programs parse their command line arguments with getopt_long, and then branch depending on those arguments; several of these branches call the same function quote(). In mkdir and mkfifo, the target is reachable within the first call to quote(), and thus SDSE can find it quickly. However, in mknod, the bug is only reachable in a later call to quote()—but since the first call to quote() is a shorter path to the target line, InterSDSE takes that call and then gets stuck inside quote(), never returning to main() to find the path to the failing assertion.

The last row in Table 1 sums up the median times for the Coreutils programs, counting time-outs as 1,800s. These results show that mixing a forward search with CCBSE can be a significant improvement—for Otter-KLEE and Random Path, the total times are notably less when mixed with CCBSE. One particularly interesting result is that Mix-CCBSE(Otter-KLEE) runs dramatically faster on mknod than either of its constituents (93.1s for the combination versus 174.8s for Otter-KLEE and 216.5s for CCBSE(RP)). This case demonstrates the benefit of mixing forward and backward search: in the combination, CCBSE(RP) found the failing path inside of quote() (recall Figure 7), and Otter-KLEE found the path from the beginning of main() to the right call to quote(). We also observe that the SIQR for Mix-CCBSE(Otter-KLEE) is generally lower than either of its constituents, which is a further benefit.

Overall, Mix-CCBSE(Otter-KLEE) has the fastest total running time across all strategies, including InterSDSE (because of its time-out); and although it is not always the fastest search strategy, it is subjectively fast enough on these examples. Thus, our results suggest that the best single strategy option for solving line reachability is Mix-CCBSE(Otter-KLEE), or perhaps Mix-CCBSE(Otter-KLEE) in round-robin with InterSDSE to combine the strengths of both.

Execution synthesis. ESD [36] is a symbolic execution tool that also aims to solve the line reachability problem. It uses a *proximity-guided path search* that is

similar to our *Intra*SDSE algorithm, and an interprocedural reaching definition analysis to find intermediate goals for directing the search. The published results show that ESD works very well on five Coreutils programs, four of which (mkdir, mkfifo, mknod, and paste) we also analyzed. Since ESD is not publicly available, we were unable to include it in our experiment directly, and we found it difficult to replicate from the description in the paper. One thing we can say for certain is that the interprocedural reaching definition analysis in ESD is clearly critical, as our implementation of IntraSDSE by itself performed quite poorly.

Comparing published numbers, if we run InterSDSE and Mix-CCBSE(Otter-KLEE) simultaneously on two machines and return whichever returns first, we obtain a strategy which performs in the same ballpark as ESD, which took 15s for mkdir, 15s for mkfifo, 20s for mknod, and 25s for paste. The ESD authors informed us that they did not observe variability in their experiment, which consists of 5 runs per test program [35]. However, we find this somewhat surprising, since ESD employs randomization in its search strategy, and is implemented on top of KLEE whose performance we have found to be highly variable (Table 1).

Clearly this comparison should be taken with a grain of salt due to major differences between Otter and ESD as well as in the experimental setups. These include the version of KLEE evaluated (we used the latest version of KLEE as of April 2011, whereas the ESD paper is based on a pre-release 2008 version of KLEE), symbolic parameters, default search strategy, processor speed, memory, Linux kernel version, whether tests are run in parallel or sequentially, the number of runs per test program, and how random number generators are seeded. These differences may also explain a discrepancy between our evaluations of KLEE: the ESD paper reported that KLEE was not able to find the target bugs within an hour, but in our experiments KLEE was able to find them (note that nearly one-third of the runs for mkdir returned within half an hour, which is not reflected by its median).

3.3 Threats to validity

There are several threats to the validity of our results. First, we were surprised by the wide variability in our running times: the SIQR can be very large—in some cases for CCBSE(*SDSE), KLEE and Otter-SAGE, the SIQR exceeds the median—and there are many outliers.[2] This indicates the results are not normally distributed, and suggests that randomness in symbolic execution can greatly perturb the results. To our knowledge, this kind of significant variability has not been reported well in the literature, and we recommend that future efforts on symbolic execution carefully consider it in their analyses. That said, the variation in results for CCBSE(Otter-KLEE) and InterSDSE, the best-performing strategies, was generally low.

Second, our implementation of KLEE and SAGE unavoidably differs from the original versions. The original KLEE is based on LLVM [22], whereas Otter

[2] See the companion technical report, Appendix A for beeswarm distribution plots for each cell in the table [23].

is based on CIL, and therefore they compute distance metrics over different control-flow graphs. Also, Otter uses newlib [28] as the standard C library, while KLEE uses uclibc [33]. These may explain some of the difference between KLEE and Otter-KLEE's performance in Table 1.

Finally, the original SAGE is a concolic executor, which runs programs to completion using the underlying operating system, while Otter-SAGE emulates the run-to-completion behavior by not switching away from the currently executing path. There are other differences between SAGE and Otter, e.g., SAGE only invokes the theorem prover at the end of path exploration, whereas Otter invokes the theorem prover at every conditional along the path. Also, SAGE suffers from *divergences*, where a generated input may not follow a predicted path (possibly repeating a previously explored path) due to mismatches between the system model and the underlying system. Otter does not suffer from divergences because it uses a purely symbolic system model. These differences may make the SAGE strategy less suited to Otter.

4 Other related work

Several other researchers have proposed general symbolic execution search strategies, in addition to the ones discussed in Section 2. Hybrid concolic testing mixes random testing with symbolic execution [24]. Burnim and Sen propose several such heuristics, including a control-flow graph based search strategy [3]. Xie et al propose Fitnex, a strategy that uses fitness values to guide path exploration [34]. It would be interesting future work to compare against these strategies as well; we conjecture that, as these are general rather than targeted search strategies, they will not perform as well as our approach for targeted search.

Researchers have also used model checkers to solve the line reachability problem by specifying the target line as the target state in the model. Much like our work, *directed model checking* [7] focuses on scheduling heuristics to quickly discover the target. Edelkamp et al proposed several heuristics based on minimizing the number of transitions from the current program state to the target state in the model defined by a finite-state automata [8] or Büchi automata [7]. Groce et al suggested using *structural heuristics* such as maximizing code coverage or thread interleavings [14]. Kupferschmid et al borrowed an AI technique based on finding the shortest distance through a *monotonic relaxation* of the model in which states are sets whose successors increase monotonically under set inclusion [20]. In contrast, SDSE prioritizes exploration based on distance in the ICFG, and CCBSE explores backwards from the target.

5 Conclusion

In this paper, we studied the problem of line reachability, which arises in automated debugging and in triaging static analysis results, among other applications. We introduced two new directed search strategies, SDSE and CCBSE, that use two very different approaches to solve line reachability. We also discussed

a method for combining CCBSE with any forward search strategy, to get the best of both worlds. We implemented these strategies and a range of state-of-the-art forward search strategies (KLEE, SAGE, and Random Path) in Otter, and studied their performance on six programs from GNU Coreutils and on two synthetic programs. The results indicate that both SDSE and mixed CCBSE and KLEE outperformed the other strategies. While SDSE performed extremely well in many cases, it does perform badly sometimes, whereas mixing CCBSE with KLEE achieves the best overall running time across all strategies, including SDSE. In summary, our results suggest that directed symbolic execution is a practical and effective approach to line reachability.

Acknowledgments

This research was supported in part by National Science Foundation grants CCF-0346982, CCF-0541036, and CCF-0915978. We would also like to thank Elnatan Reisner and Jonathan Turpie for their help developing the POSIX library.

References

1. Richard Bornat. Proving pointer programs in Hoare logic. In *MPC*, pages 102–126, 2000.
2. Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT–a formal system for testing and debugging programs by symbolic execution. In *ICRS*, pages 234–245, 1975.
3. Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *ASE*, pages 443–446, 2008.
4. Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.
5. Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. In *CCS*, pages 322–335, 2006.
6. Coreutils - GNU core utilities. http://www.gnu.org/software/coreutils/.
7. Stefan Edelkamp, Stefan Leue, and Alberto Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *Software Tools for Technology Transfer*, 5(2):247–267, 2004.
8. Stefan Edelkamp, Alberto Lluch-Lafuente, and Stefan Leue. Trail-directed model checking. *Electrical Notes Theoretical Computer Science*, 55(3):343–356, 2001.
9. Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable context-sensitive flow analysis using instantiation constraints. In *PLDI*, pages 253–263, 2000.
10. Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *CAV*, pages 519–531, 2007.
11. Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *PLDI*, pages 213–223, 2005.
12. Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Active property checking. In *EMSOFT*, pages 207–216, 2008.

13. Patrice Godefroid, Michael Y. Levin, and David A Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.
14. Alex Groce and Willem Visser. Model checking Java programs using structural heuristics. In *ISSTA*, pages 12–21, 2002.
15. William E. Howden. Symbolic testing and the DISSECT symbolic evaluation system. *IEEE Transactions on Software Engineering*, 3(4):266–278, 1977.
16. Yit Phang Khoo, Bor-Yuh Evan Chang, and Jeffrey S. Foster. Mixing type checking and symbolic execution. In *PLDI*, pages 436–447, 2010.
17. James C. King. Symbolic execution and program testing. *CACM*, 19(7):385–394, 1976.
18. The KLEE Symbolic Virtual Machine. http://klee.llvm.org.
19. John Kodumal and Alex Aiken. The set constraint/CFL reachability connection in practice. In *PLDI*, pages 207–218, 2004.
20. Sebastian Kupferschmid, Jörg Hoffmann, Henning Dierks, and Gerd Behrmann. Adapting an AI planning heuristic for directed model checking. In Antti Valmari, editor, *SPIN*, volume 3925 of *LNCS*, pages 35–52. Springer Berlin / Heidelberg, 2006.
21. William Landi and Barbara G. Ryder. Pointer-induced aliasing: a problem taxonomy. In *POPL*, pages 93–103, 1991.
22. Chris Lattner and Vikram Adve. LLVM: a compilation framework for lifelong program analysis transformation. In *CGO*, pages 75–86, 2004.
23. Kin-Keung Ma, Yit Phang Khoo, Jeffrey S. Foster, and Michael Hicks. Directed symbolic execution. Technical Report CS-TR-4979, UMD-College Park, Apr 2011.
24. Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *ICSE*, pages 416–426, 2007.
25. Jim Meyering. seq: give a proper diagnostic for an invalid –format=% option, 2008. http://git.savannah.gnu.org/cgit/coreutils.git/commit/?id=b8108fd2ddf77ae79cd014f4f37798a52be13fd1.
26. Joe M. Morris. A general axiom of assignment. Assignment and linked data structure. A proof of the Schorr-Waite algorithm. In M Broy and G. Schmidt, editors, *Theoretical Foundations of Programming Methodology*, pages 25–51, 1982.
27. George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC*, pages 213–228, 2002.
28. The Newlib Homepage. http://sourceware.org/newlib/.
29. Leon J. Osterweil and Lloyd D. Fosdick. Program testing techniques using simulated execution. In *ANSS*, pages 171–177, 1976.
30. Jakob Rehof and Manuel Fähndrich. Type-base flow analysis: from polymorphic subtyping to CFL-reachability. In *PLDI*, pages 54–66, 2001.
31. Elnatan Reisner, Charles Song, Kin-Keun Ma, Jeffrey S. Foster, and Adam Porter. Using symbolic evaluation to understand behavior in configurable software systems. In *ICSE*, pages 445–454, 2010.
32. Thomas W. Reps. Program analysis via graph reachability. In *ILPS*, pages 5–19, 1997.
33. μClibc. http://www.uclibc.org/.
34. Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *DSN*, pages 359–368, 2009.
35. Cristian Zamfir. Personal communication, May 2011.
36. Cristian Zamfir and George Candea. Execution synthesis: a technique for automated software debugging. In *EuroSys*, pages 321–334, 2010.