Directed symbolic execution

Kin-Keung Ma, <u>Khoo</u> 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, e.g., a developer might learn that a failure is possible on a particular line but might not know exactly how to reproduce the failure or even whether it is reproducible. This can happen particularly often for bug reports from static analysis tools, which can produce false positives. 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 Mix-CCBSE, a strategy in which CCBSE is composed with another search strategy to yield a hybrid that is more powerful than either strategy alone. We compare SDSE, CCBSE, and Mix-CCBSE with several existing strategies from the literature. We find that, while SDSE performs extremely well in many cases, it sometimes fails badly. However, by mixing CCBSE with KLEE's search strategy, we obtain a strategy that has the best overall performance across the strategies we studied.

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 [16]. 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.

In this paper, we explore using symbolic execution to solve the line reachability problem. Symbolic executors work by running the program, computing over both concrete values and expressions that include *symbolic variables*, which are unknowns that range over various sets of values, e.g., integers, strings, etc. [21, 3, 18, 32]. When a symbolic executor encounters a conditional whose guard depends on a symbolic variable, it invokes a theorem prover (e.g., an SMT solver such as Z3 [8], Yices [9], or STP [12]) to determine which branches are feasible. If both are, the symbolic execution conceptually forks, exploring both branches.

Symbolic execution is an attractive approach to solving line reachability, because it is *complete*, meaning any path it finds is realizable, and the symbolic executor can even construct a concrete test case that takes that path (using the SMT solver). However, symbolic executors cannot explore all program paths, and hence must make heuristic choices about which paths to take. There is a significant body of work exploring such heuristics [15, 6, 5, 26, 4, 41], but the focus of most prior work is on increasing symbolic execution's useful coverage in general, rather than trying to reach certain lines in particular, as is the goal of the current work. We are aware of one previously proposed approach, *execution synthesis* [42], for using symbolic execution to solve the line reachability problem; we discuss execution synthesis in conjunction with our experiments in Section 4.

We propose two new *directed* symbolic execution search strategies for line reachability. First, we propose *shortest-distance symbolic execution (SDSE)*, which guides symbolic execution using distance computed in an interprocedural control-flow graph (ICFG). More specifically, when the symbolic executor is deciding which path to continue executing, it will pick the path that currently has the shortest distance to the target line in the ICFG. SDSE is inspired by a heuristic used in the coverage-based search strategy from KLEE [5], but adapted to focus on a single target line, rather than all uncovered lines.

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 symbolic execution as a subroutine. More specifically, suppose target line l is inside function f. CCBSE begins forward symbolic execution from the start of f, looking for paths to l. Since CCBSE does not know how f is called, it sets all of f's inputs (parameters and global variables) to be purely symbolic. This initial execution yields a set of partial paths $\bar{p}$ that start at f and lead to l; in a sense, these partial paths summarize selected behavior of f. Next, CCBSE finds possible callers of f, and for each such caller g, CCBSE runs forward symbolic execution from its start (again setting g's inputs symbolic), now searching for paths that call f. For each such path p, it attempts to continue down paths p' in $\bar{p}$ until reaching l. For those extended paths $p + p'$ that are feasible, it adds them to $\bar{p}$. (An extended path may be infeasible because the precondition needed for p' may not be satisfied by p.) This process continues, working backward up the call chain until CCBSE finds a path from the start of the program to l, or the process times out.

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 working with a constraint language that explicitly summarizes 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 chances 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 [36]. We also extended Otter with two popular forward search strategies from KLEE [5] and SAGE [17]; for a baseline, we also include 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 [7], 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, while SDSE performs extremely well on many programs, it can fall short very badly under certain program patterns. By mixing CCBSE with KLEE, we obtain a strategy that has the best overall performance across all strategies. Our results suggest that directed symbolic execution is a practical and effective new approach to solving the line reachability problem.

2 Overview and motivating examples

In this section we present an overview of our ideas—SDSE, CCBSE, and Mix-CCBSE—for directed symbolic execution. We will refer to Otter, our symbolic execution framework, to make our explanations concrete (and to save space), but the basic ideas apply to any symbolic execution tool [21, 15, 5, 19].

Figure 1 diagrams the architecture of Otter and gives pseudocode for its main scheduling loop. Otter uses CIL [30] to produce a control-flow graph from the
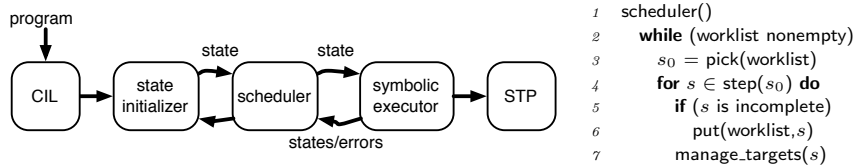
```
1   scheduler()
2     while (worklist nonempty)
3       s_0 = pick(worklist)
4       for s ∈ step(s_0) do
5         if (s is incomplete)
6           put(worklist,s)
7       manage_targets(s)
```

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

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 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 variables*. At these points, Otter queries STP [12], an SMT solver, to see if legal valuations of the symbolic variables 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 guides CCBSE's backward search, and is discussed further in Section 3.2.
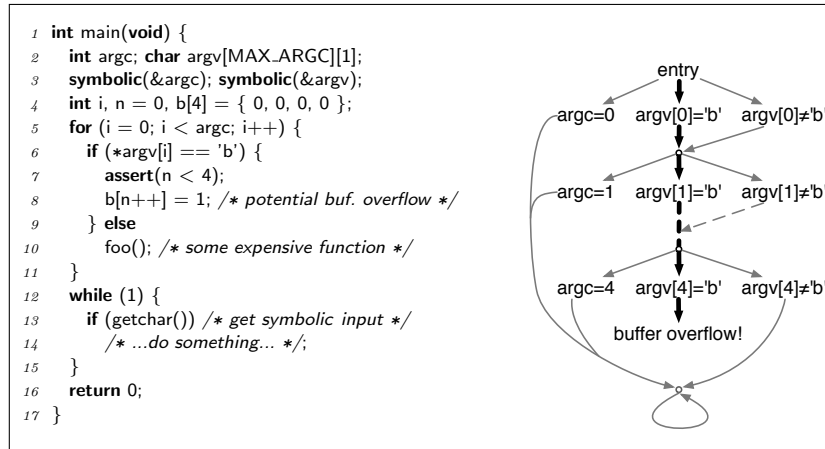
2.1 Forward symbolic execution

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

Random Path (RP) [5] 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 [5] 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 [22].

SAGE [17] 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,

```
 1  int main(void) {
 2    int argc; char argv[MAX_ARGC][1];
 3    symbolic(&argc); symbolic(&argv);
 4    int i, n = 0, b[4] = { 0, 0, 0, 0 };
 5    for (i = 0; i < argc; i++) {
 6      if (*argv[i] == 'b') {
 7        assert(n < 4);
 8        b[n++] = 1; /* potential buf. overflow */
 9      } else
10        foo(); /* some expensive function */
11    }
12    while (1) {
13      if (getchar()) /* get symbolic input */
14        /* ...do something... */;
15    }
16    return 0;
17  }
```

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

grandchildren of the first generation, etc) in a more intermixed fashion, using a block coverage heuristic to determine which generations to explore first.

2.2 Shortest-distance symbolic execution

To see 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 ith 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 ensure this overflow does not occur. After the arguments are processed, the program enters a loop that reads and processes character inputs (lines 12 and below).

Suppose we would like to reason about a possible failure of the assertion. Then we can run this program with symbolic inputs, which we achieve 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 1 path that reaches the failing assertion out of $\sum_{n=0}^{4} 3 \times 2^n = 93$ total paths. Moreover, the assertion is no longer reachable once exploration has advanced past the argument-processing loop.

In this example, RP would have only a small chance of finding the overflow, since it will be biased towards exploring edges before 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
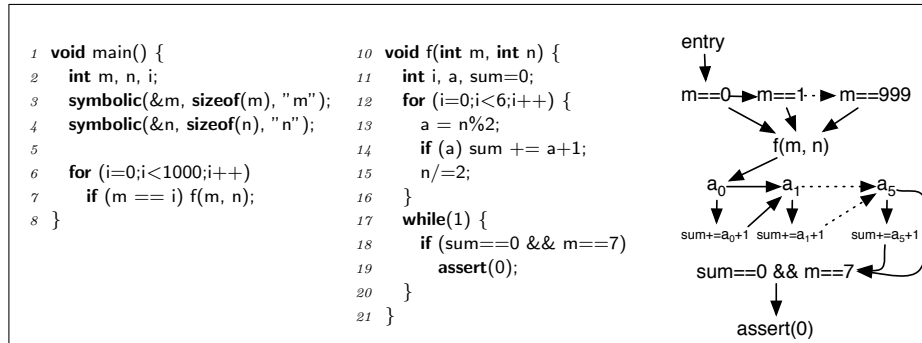
```
 1  void main() {
 2     int m, n, i;
 3     symbolic(&m, sizeof(m), "m");
 4     symbolic(&n, sizeof(n), "n");
 5
 6     for (i=0;i<1000;i++)
 7        if (m == i) f(m, n);
 8  }
```

```
10  void f(int m, int n) {
11     int i, a, sum=0;
12     for (i=0;i<6;i++) {
13        a = n%2;
14        if (a) sum += a+1;
15        n/=2;
16     }
17     while(1) {
18        if (sum==0 && m==7)
19           assert(0);
20     }
21  }
```

Fig. 3. Example illustrating CCBSE's potential benefit.

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 instruction 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.

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.

Consider the code in Figure 3. This program initializes m and n to be symbolic and then loops, calling f(m, n) when m == i for $i \in [0, 1000)$. 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.
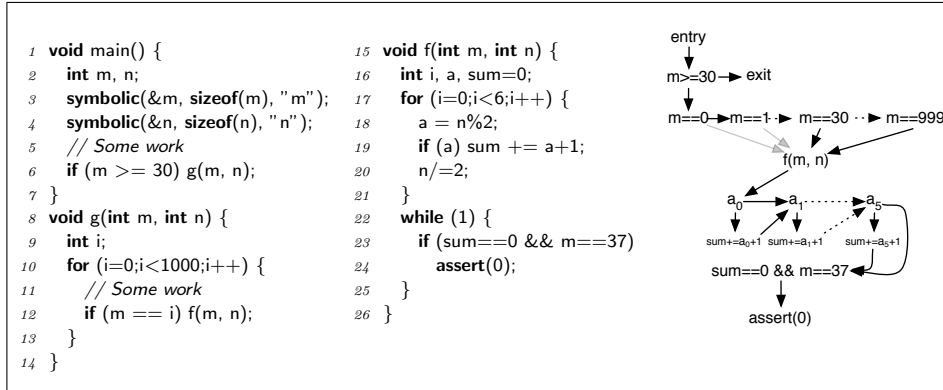
```
 1   void main() {
 2     int m, n;
 3     symbolic(&m, sizeof(m), "m");
 4     symbolic(&n, sizeof(n), "n");
 5     // Some work
 6     if (m >= 30) g(m, n);
 7   }
 8   void g(int m, int n) {
 9     int i;
10     for (i=0;i<1000;i++) {
11       // Some work
12       if (m == i) f(m, n);
13     }
14   }
```

```
15   void f(int m, int n) {
16     int i, a, sum=0;
17     for (i=0;i<6;i++) {
18       a = n%2;
19       if (a) sum += a+1;
20       n/=2;
21     }
22     while (1) {
23       if (sum==0 && m==37)
24         assert(0);
25     }
26   }
```

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

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 explore all 2^6 paths induced by the for loop, and one of them, say p, will reach the assertion. Since m is purely symbolic, m == 7 is satisfiable. Then inside main, CCBSE explores various paths that reach the call to f and then try to follow p; hence CCBSE is able to short-circuit evaluation inside of f (in particular, the 2^6 branching induced by the for-loop), and thus quickly find a realizable path to the failure.

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 4, a modified version of the code in Figure 3. Here, main calls function g, which acts as main did in Figure 3, with some m >= 30 (line 6), and the assertion in f is reachable only when m == 37 (line 23). All other strategies fail in the same manner as they do in Figure 3.

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. The backward search, just like CCBSE, finds a path from f's entry to the assertion, but then gets stuck in finding a path from g's entry to the same 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.
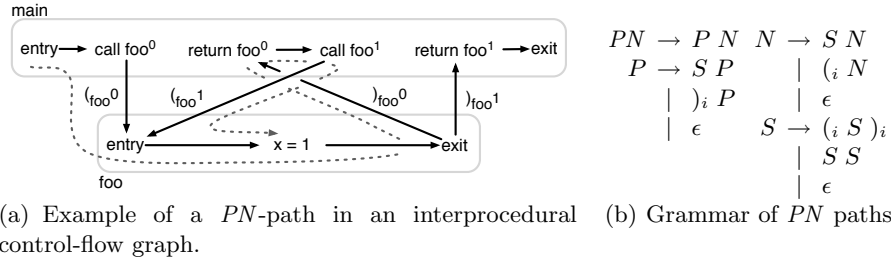
(a) Example of a *PN*-path in an interprocedural control-flow graph.

(b) Grammar of *PN* paths.

Fig. 5. SDSE distance computation.

3 Directed symbolic execution

In this section we present the details of our implementations of SDSE, CCBSE, and Mix-CCBSE in Otter.

3.1 Shortest-distance symbolic execution

SDSE is implemented as a pick function from Figure 1. As previously described, SDSE chooses the state on the worklist with the shortest *distance to target*. We compute distances over an interprocedural control-flow graph (ICFG) [24], 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 5(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* grammar shown in Figure 5(b). In this grammar, developed by Fähndrich et al [11, 35], *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 5(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 [11, 35, 23]. 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 initial *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.

3.2 Call-chain-backward symbolic execution

CCBSE is implemented in the `manage_targets` and `pick` functions from Figure 1. Otter classifies states s according to the function f in which symbolic execution starts, which we call the *origin function*. Thus, traditional symbolic execution states always have `main` as their origin function, while CCBSE requires additional origin functions. In particular, CCBSE begins by initializing states for functions containing target lines.

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`. At the start of CCBSE with a single target, there will be only one function to choose, but as execution continues there will be more choices, as we show below, and "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 using a user-selected forward search strategy. We write CCBSE(S) to denote CCBSE using forward search strategy S.

The `manage_targets`(s) function is given in Figure 6. 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 the path's end (denoted `pc(`p`)`) has reached a target, 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`. First, we simply add p to `f`'s paths (line 17). If `f` did not yet have any paths, it will create initial states for each of `f`'s callers (pre-computed from the call graph) and add these to the worklist (line 18). Because these callers will be closer to `main`, they will be subsequently favored by `pick` when it chooses states.

We implement Mix-CCBSE with a slight alteration to `pick` as described above. At each step, we decide whether to use CCBSE next, or whether to use regular forward search. The decision is made based on the *system time* (computed as $50\times$ (no. of solver calls) + (no. of instructions executed)) previously spent by each strategy: if CCBSE has spent less system time in the past than forward search, then it gets to run next, and otherwise the forward strategy runs next. This approach splits the time between CCBSE and forward search roughly 50/50, with variation due to differences between the time various solver queries or instruction

```
 8  manage_targets (s)                    16  update_paths (f, p)
 9    (sf,p) = path(s)                     17    add_path(f, p);
10    if pc(p) ∈ targets                   18    if not(has_paths(f))
11      update_paths(sf, p)                19      add_callers(f,worklist)
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′)
```

Fig. 6. Target management for CCBSE.

executions take.[1] We could also allow the user to vary this percentage, as desired. We write Mix-CCBSE(S_F,S_B) to denote the mixed strategy where S_F is the forward search strategy and CCBSE(S_B) is the backward strategy.

3.3 Starting symbolic execution "in the middle"

As mentioned above, the state initializer for CCBSE may start execution at an arbitrary function, i.e., "in the middle" of an execution. Thus, it must generate symbolic values for function inputs (parameters and global variables) that represent all possible initial conditions the function may be called in. We can initialize integer-valued data to purely symbolic words, but representing arbitrarily complex, pointer-ful data structures presents additional challenges. For example, we may start at a function that expects a linked list as its input, and thus we want to create a symbolic state that can represent linked lists of any length.

In Otter, memory is represented as a map from abstract addresses to symbolic values, and each variable or dynamic memory allocation is given a separate mapping in memory. Pointers are represented as ⟨*base*, *offset*⟩ pairs, where *base* is the index into the memory map, and *offset* is a concrete or symbolic integer (indicating the position of a struct field or array element). Null pointers are represented as the integer 0. Additionally, we represent pointers that may point to multiple base addresses using *conditional pointers* of the form *if g then p else q*, where g is a boolean symbolic value while p and q are conditionals or base-offset pairs. For example, a pointer that may be null, point to a variable x, or point to the second element of an array y could be represented as *if g_0 then* 0 *else if g_1 then* ⟨&x, 0⟩ *else* ⟨&y, 2⟩. Reads and writes through conditional pointers use Morris's general axiom of assignment [2, 29].

Thus, for pointer-valued inputs to functions at which CCBSE starts execution, we can represent arbitrary initial aliasing using conditional pointers in

---

[1] We could also split execution in terms of wall-clock time spent in each strategy. However, this leads to non-deterministic, non-reproducible results. We opted to use our notion of system time for reproducibility.

which the guards g are fresh symbolic boolean variables. We experimented with using a sound pointer analysis to seed this process, but found that it was both expensive (the conditional pointers created were complex, leading to longer solving times) and unnecessary in practice. Instead, we use an unsound heuristic: 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 to be a limitation. Note these choices in modeling pointers only mean that CCBSE could miss some targets, which any heuristic symbolic execution search could also do; because CCBSE always works backward to the start of main, all the final paths it produces are still feasible.

To be able to represent 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.

4 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 [5].

The results are presented in Table 1. Part (a) of the table gives results for our directed search strategies. For comparison purposes, we also implemented an intraprocedural variant of SDSE; we refer to that variant as IntraSDSE, and to the strategy from Section 3.1 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 [22], 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). Since we always mix strategies with CCBSE(RP), we will write Mix-CCBSE(S) as an abbreviation for Mix-CCBSE(S, RP). We did not directly compare against execution synthesis (ESD) [42], 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

	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 3	∞	∞	60.8 7.8(4)	7.3 1.2(3)	7.2 1.0(4)	∞
Figure 4	∞	∞	∞	∞	∞	∞
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 3	579.7 ∞	205.5 133.1(9)	∞	∞	160.1 6.4(11)	80.6 177.2(9)
Figure 4	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.   Key: | Median SIQR(Outliers) | ∞ : time out

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). In parentheses are the number of outliers, which fall 3×SIQR below the lower quartile or above the upper quartile, if non-zero. We ran each test for a maximum of 600 seconds for the synthetic examples, and 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. The fastest two times in each row are highlighted.

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 only 16 tests in parallel, to minimize resource contention. The results required less than 2 days of elapsed time. Total memory usage was below 1GB per test.

4.1 Synthetic programs

The first two rows in Table 1 give the results from the examples in Figures 2, 3, and 4. 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 3, 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 $0 \leq m < 7$), 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 4, 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.

4.2 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 [28]. Thus, just before this array access, we added an assertion **assert**(i<strlen(fmt)) to indicate the overflow. Note that Otter does have built-in detection of buffer overflows and similar errors, but for this experiment we do not count those as 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 [5]: 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 implemented as an unbounded stream of symbolic values.

Since Otter is a source-code-only symbolic executor, we needed complete library function source code to symbolically execute Coreutils programs. To this end, we implemented a symbolic model of POSIX system calls and the file system, and use newlib [31] as the C standard library.

```
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

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 [42] is a symbolic execution tool that also aims to solve the line reachability problem (ESD also includes other features, such as support for multi-threaded programs). Given a program and a bug report, ESD extracts the bug location and tries to produce an execution that leads to the bug. It uses a *proximity-guided path search* that is similar to our *Intra*SDSE algorithm. ESD also uses an interprocedural reaching definition analysis to find intermediate goals from the program start to the bug location, and directs search to those

goals. 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, InterSDSE in parallel with Mix-CCBSE(Otter-KLEE) performs in the same ballpark as ESD, which took 15s for mkdir, 15s for mkfifo, 20s for mknod, and 25s for paste. The authors informed us that they did not observe variability in their experiment, which consists of 5 runs per test program [43]. However, we find this surprising, since ESD employs randomization in its search strategy, and is implemented on top of KLEE which 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 those bugs within an hour, but we were able to find them with KLEE (nearly one-third of the runs for mkdir returned within half an hour, which is not reflected by its median; see Appendix A.1).

Threats to validity. There are several threats to the validity of our results. First, we were surprised by the wide variability in our results: the SIQR can be very large—in some cases for CCBSE(*SDSE) and Otter-SAGE, the SIQR exceeds the median—and there are some 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 [25], whereas Otter is based on CIL, and therefore they compute distance metrics over different control-flow graphs. Also, Otter uses newlib [31] as the standard C library, while KLEE uses uclibc [39]. These may explain some of the difference between KLEE and Otter-KLEE's performance in Table 1.

---

[2] Appendix A shows beeswarm distribution plots for each cell in the results table.

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.

5   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 [26]. Burnim and Sen propose several such heuristics, including a control-flow graph based search strategy [4]. Xie et al propose Fitnex, a strategy that uses fitness values to guide path exploration [41]. 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.

In addition to improved search strategies, researchers have explored other means to make symbolic execution more scalable. Majumdar and Xu propose using symbolic grammars to guide symbolic execution by reducing the space of possible inputs [27]. Godefroid et al propose a similar idea with improved scalability [14]. Boonstoppel et al propose RWset analysis to prune symbolic execution paths that are redundant with prior path exploration [1]. Compositional Dynamic Testing creates function summaries during symbolic execution to improve performance [13]. Loop-extended symbolic execution attempts to summarize multiple loop iterations, so that a small number of symbolic executions of a loop body can capture many possible unrollings of the loop [37]. While all of these ideas are promising, they are orthogonal to the directed symbolic execution we explore in this paper—it should be possible to combine any of them with our proposed search strategies.

Prior work has explored the issue of initializing pointers in symbolic execution. Symbolic Java PathFinder lazily initializes object references, and uses types to infer aliasing [20]. CUTE, a concolic executor, starts at an arbitrary function by initializing pointers based at first on a simple heap with abstract addresses, and incrementally increasing the heap complexity in subsequent runs [38]. Extensions to SAGE [10] and to Pex [40] build on CUTE by modeling pointers as symbolic integers; the latter also uses types to enforce non-aliasing constraints. Otter combines the approaches of JPF and SAGE/Pex by modeling pointers as lazily initialized conditionals of abstract addresses with symbolic integer offsets.

6 Conclusion

In this paper, we studied the problem of line reachability, which arises in the applications of automated debugging and triaging static analysis results. 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.

References

1. P. Boonstoppel, C. Cadar, and D. Engler. RWset: Attacking path explosion in constraint-based test generation. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 351–366, 2008.
2. Richard Bornat. Proving pointer programs in Hoare logic. In *International Conference on Mathematics of Program Construction (MPC)*, pages 102–126, London, UK, 2000. Springer-Verlag.
3. R.S. Boyer, B. Elspas, and K.N. Levitt. SELECT–a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the international conference on Reliable software*, pages 234–245. ACM, 1975.
4. Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *ASE*, pages 443–446. IEEE, 2008.
5. 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.
6. 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.
7. Coreutils - GNU core utilities, 2011. http://www.gnu.org/software/coreutils/.
8. Leonardo de Moura and Nikolaj Bjørner. Z3: Efficient SMT solver. In *TACAS'08: Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963/2008 of *Lecture Notes in Computer Science*, pages 337–340, 2008.
9. B. Dutertre and L. De Moura. A fast linear-arithmetic solver for DPLL (T). In *Computer Aided Verification*, pages 81–94. Springer, 2006.
10. Bassem Elkarablieh, Patrice Godefroid, and Michael Y. Levin. Precise pointer reasoning for dynamic test generation. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA '09, pages 129–140, New York, NY, USA, 2009. ACM.
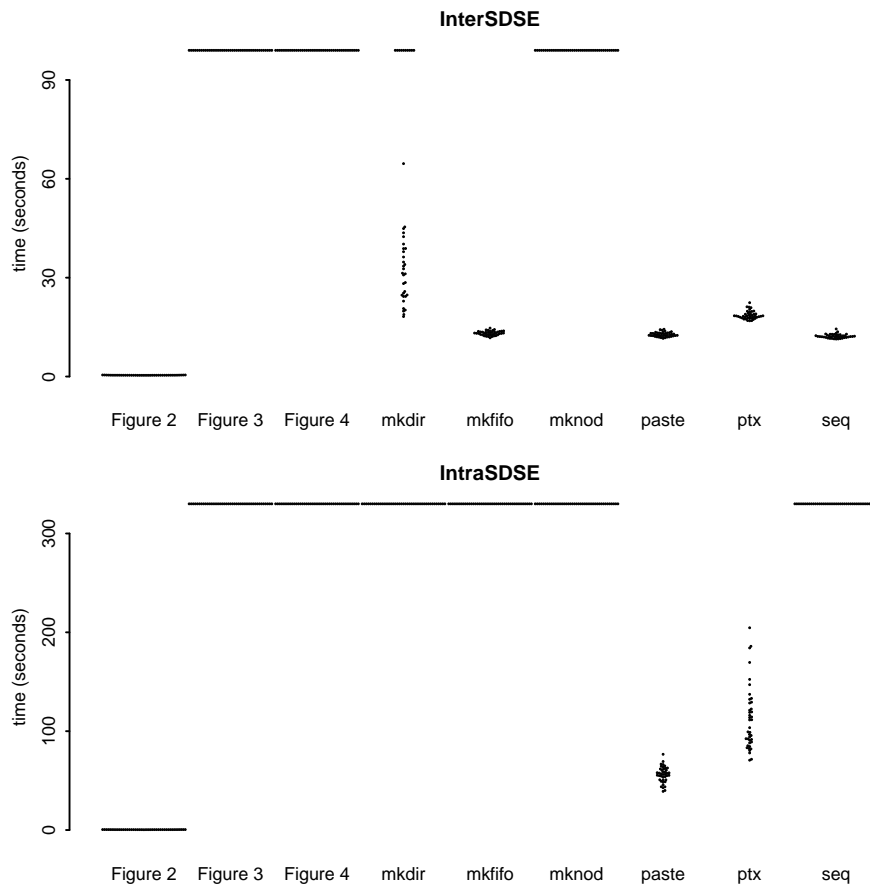
11. Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 253–263, New York, NY, USA, 2000. ACM.

12. V. Ganesh and D. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification*, pages 519–531. Springer, 2007.

13. Patrice Godefroid. Compositional dynamic test generation. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '07, pages 47–54, New York, NY, USA, 2007. ACM.

14. Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *PLDI*, pages 206–215, 2008.

15. Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *PLDI*, pages 213–223, 2005.

16. Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Active property checking. In *Proceedings of the 8th ACM international conference on Embedded software*, 2008.

17. Patrice Godefroid, Michael Y. Levin, and David A Molnar. Automated whitebox fuzz testing. In *NDSS*. Internet Society, 2008.

18. W.E. Howden. Symbolic testing and the DISSECT symbolic evaluation system. *IEEE Transactions on Software Engineering*, pages 266–278, 1977.

19. Yit Phang Khoo, Bor-Yuh Evan Chang, and Jeffrey S. Foster. Mixing Type Checking and Symbolic Execution. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 436–447, Toronto, Canada, June 2010.

20. Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the 9th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'03, pages 553–568, Berlin, Heidelberg, 2003. Springer-Verlag.

21. James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

22. The KLEE Symbolic Virtual Machine. http://klee.llvm.org.

23. John Kodumal and Alex Aiken. The set constraint/cfl reachability connection in practice. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 207–218, New York, NY, USA, 2004. ACM.

24. William Landi and Barbara G. Ryder. Pointer-induced aliasing: a problem taxonomy. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '91, pages 93–103, New York, NY, USA, 1991. ACM.

25. C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75 – 86, march 2004.

26. Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 416–426, Washington, DC, USA, 2007. IEEE Computer Society.

27. Rupak Majumdar and Ru-Gang Xu. Directed test generation using symbolic grammars. In *ASE*, pages 134–143, 2007.

28. Jim Meyering. seq: give a proper diagnostic for an invalid –format=% option, 2008. http://git.savannah.gnu.org/cgit/coreutils.git/commit/?id=b8108fd2ddf77ae79cd014f4f37798a52be13fd1.
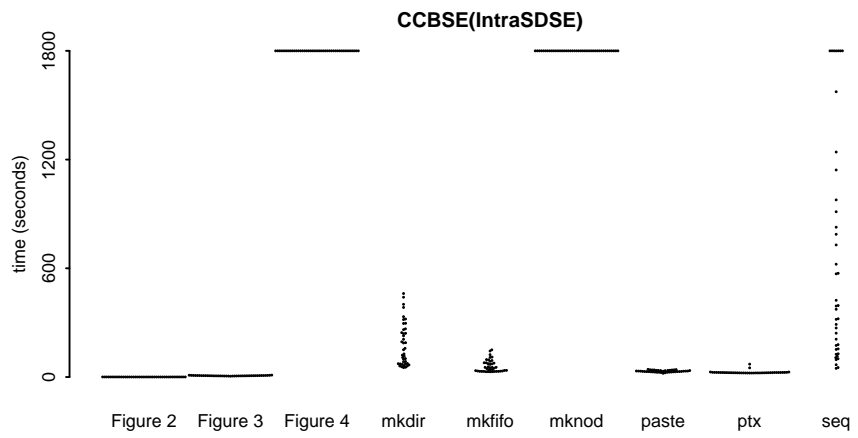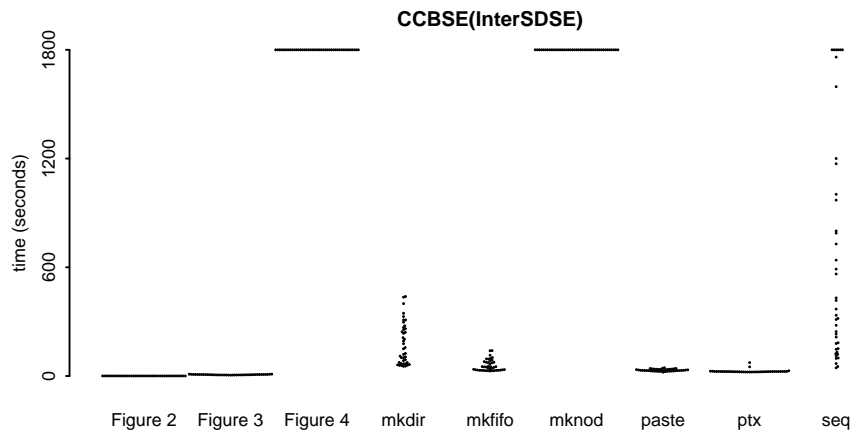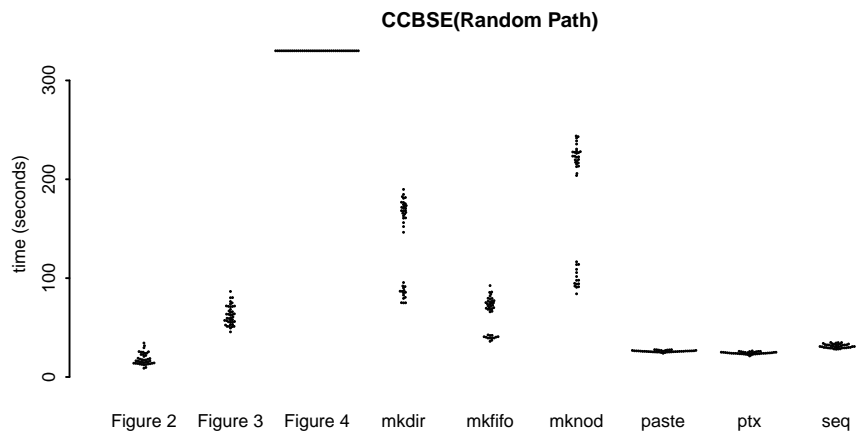
29. Joe M. Morris. A general axiom of assignment. Assignment and linked data structure. A proof of the Schorr-Waite algorithm. In M Broy and G. Schmidt, editors, *Theoretical Foundations of Programming Methodology*, pages 25–51. Reidel, 1982.

30. George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Conference on Compiler Construction (CC)*, pages 213–228, London, UK, 2002. Springer-Verlag.

31. The Newlib Homepage, 2011. http://sourceware.org/newlib/.

32. L.J. Osterweil and L.D. Fosdick. Program testing techniques using simulated execution. In *Proceedings of the 4th symposium on Simulation of computer systems*, pages 171–177. IEEE Press, 1976.

33. The R Project for Statistical Computing, 2011. http://www.r-project.org/.

34. beeswarm: an R package, 2011. http://www.cbs.dtu.dk/ eklund/beeswarm/.

35. Jakob Rehof and Manuel Fähndrich. Type-base flow analysis: from polymorphic subtyping to cfl-reachability. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '01, pages 54–66, New York, NY, USA, 2001. ACM.

36. Elnatan Reisner, Charles Song, Kin-Keun Ma, Jeffrey S. Foster, and Adam Porter. Using Symbolic Evaluation to Understand Behavior in Configurable Software Systems. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, pages 445–454, Cape Town, South Africa, May 2010.

37. P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended symbolic execution on binary programs. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 225–236. ACM, 2009.

38. Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *FSE-13*, pages 263–272, 2005.

39. μClibc, 2011. http://www.uclibc.org/.

40. Dries Vanoverberghe, Nikolai Tillmann, and Frank Piessens. Test input generation for programs with pointers. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,*, TACAS '09, pages 277–291, Berlin, Heidelberg, 2009. Springer-Verlag.

41. T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*, pages 359–368. IEEE, 2009.

42. C. Zamfir and G. Candea. Execution synthesis: a technique for automated software debugging. In *Proceedings of the 5th European conference on Computer systems*, pages 321–334. ACM, 2010.

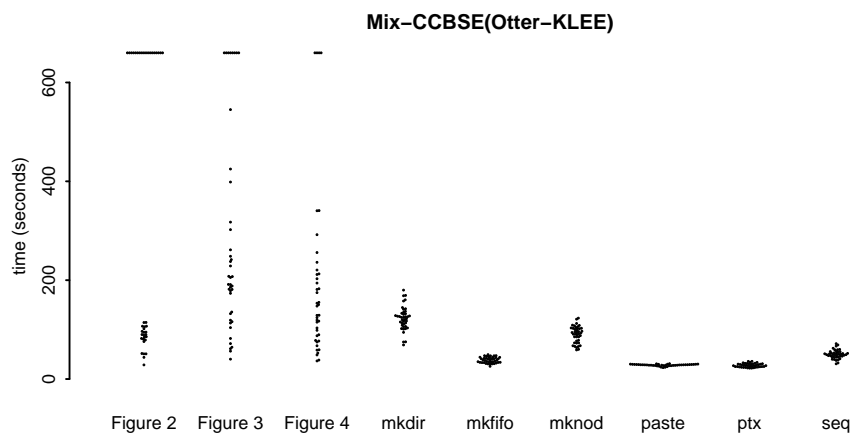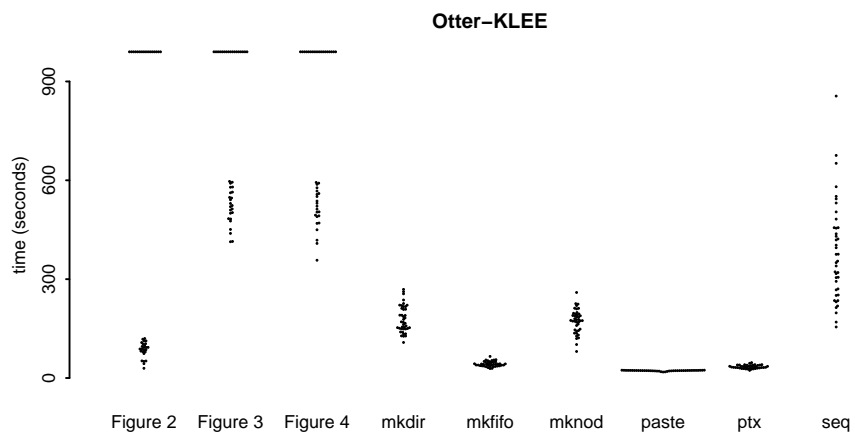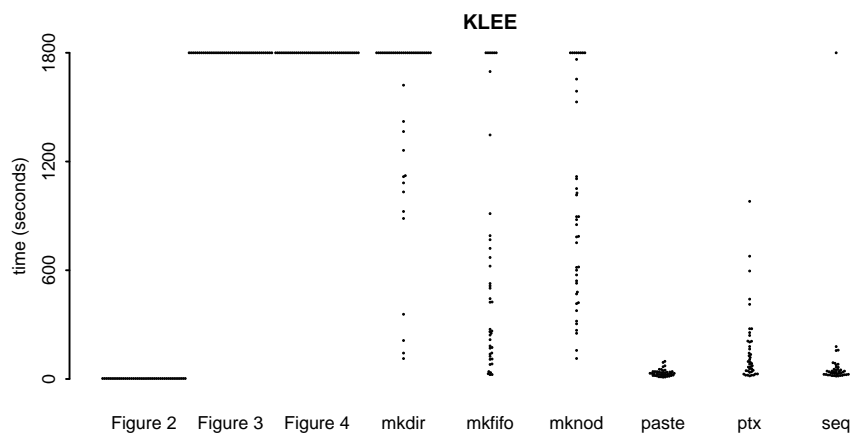43. Cristian Zamfir. Personal communication, May 2011.
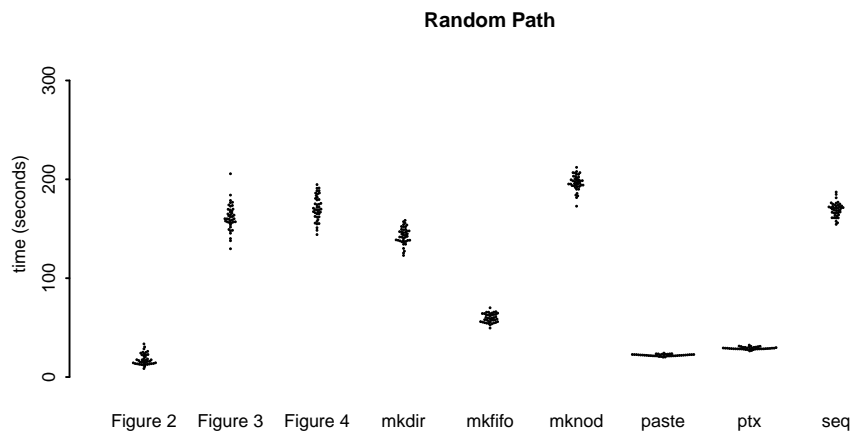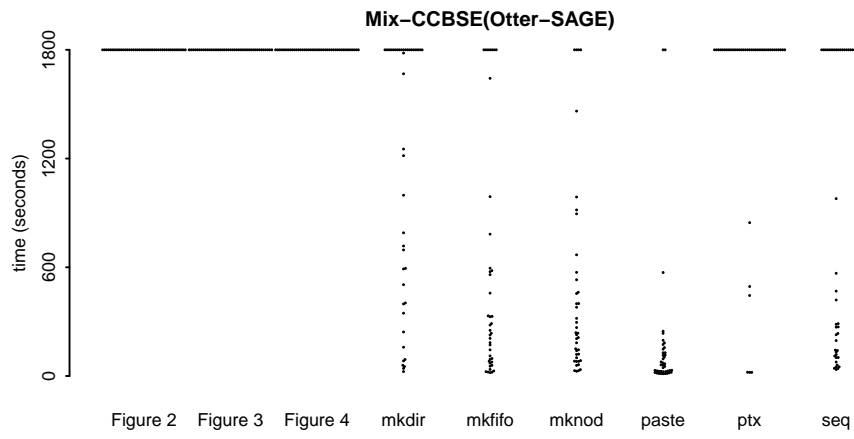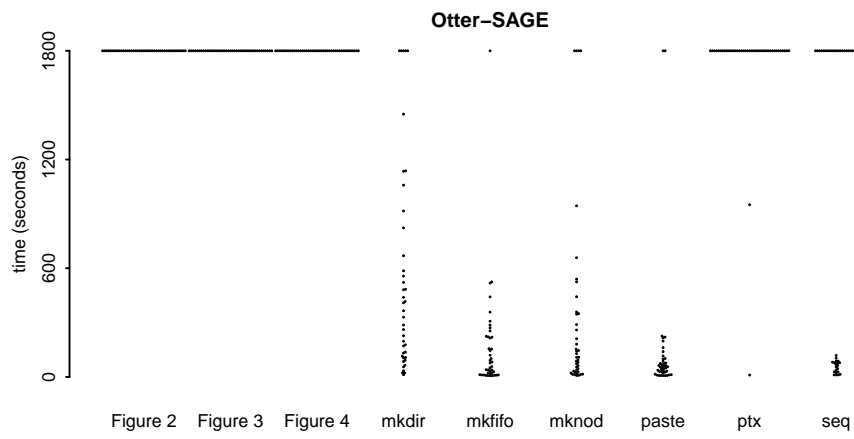
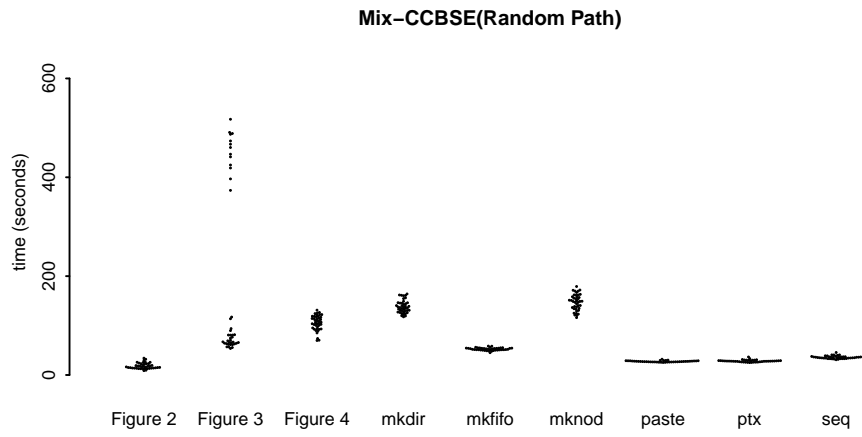# A   Beeswarm distribution plots of benchmark results

A.1   Grouped by strategy

The following plots are beeswarm distribution plots generated in R [33] using the beeswarm [34] package. Each set of plots corresponds to a strategy, and each subplot to a benchmark program from our experiment (Section 4). Each point corresponds to the time it takes for a single run to complete. The points are plotted vertically along the y axis, which is scaled to the slowest run that did not time out for each strategy across all benchmark programs, and randomly dispersed horizontally to avoid overlap. Runs that timed out are plotted either just above the upper limit of y axis, or at 1,800 seconds.
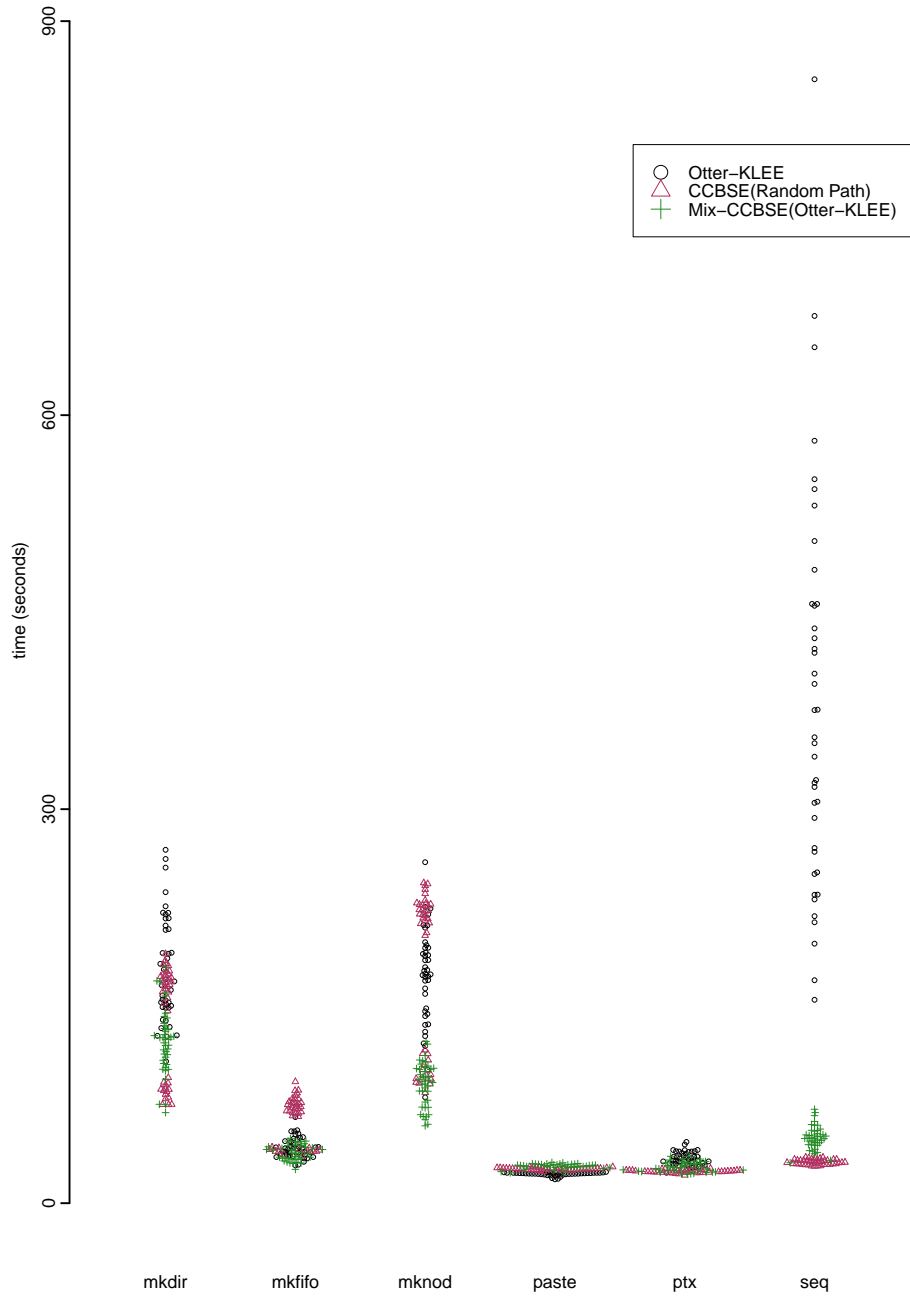
CCBSE(Random Path)

time (seconds)

300
200
100
0

Figure 2 Figure 3 Figure 4 mkdir mkfifo mknod paste ptx seq

CCBSE(InterSDSE)

time (seconds)

1800
1200
600
0

Figure 2 Figure 3 Figure 4 mkdir mkfifo mknod paste ptx seq

CCBSE(IntraSDSE)

time (seconds)

1800
1200
600
0

Figure 2 Figure 3 Figure 4 mkdir mkfifo mknod paste ptx seq

KLEE

Otter−KLEE

Mix−CCBSE(Otter−KLEE)

Otter–SAGE

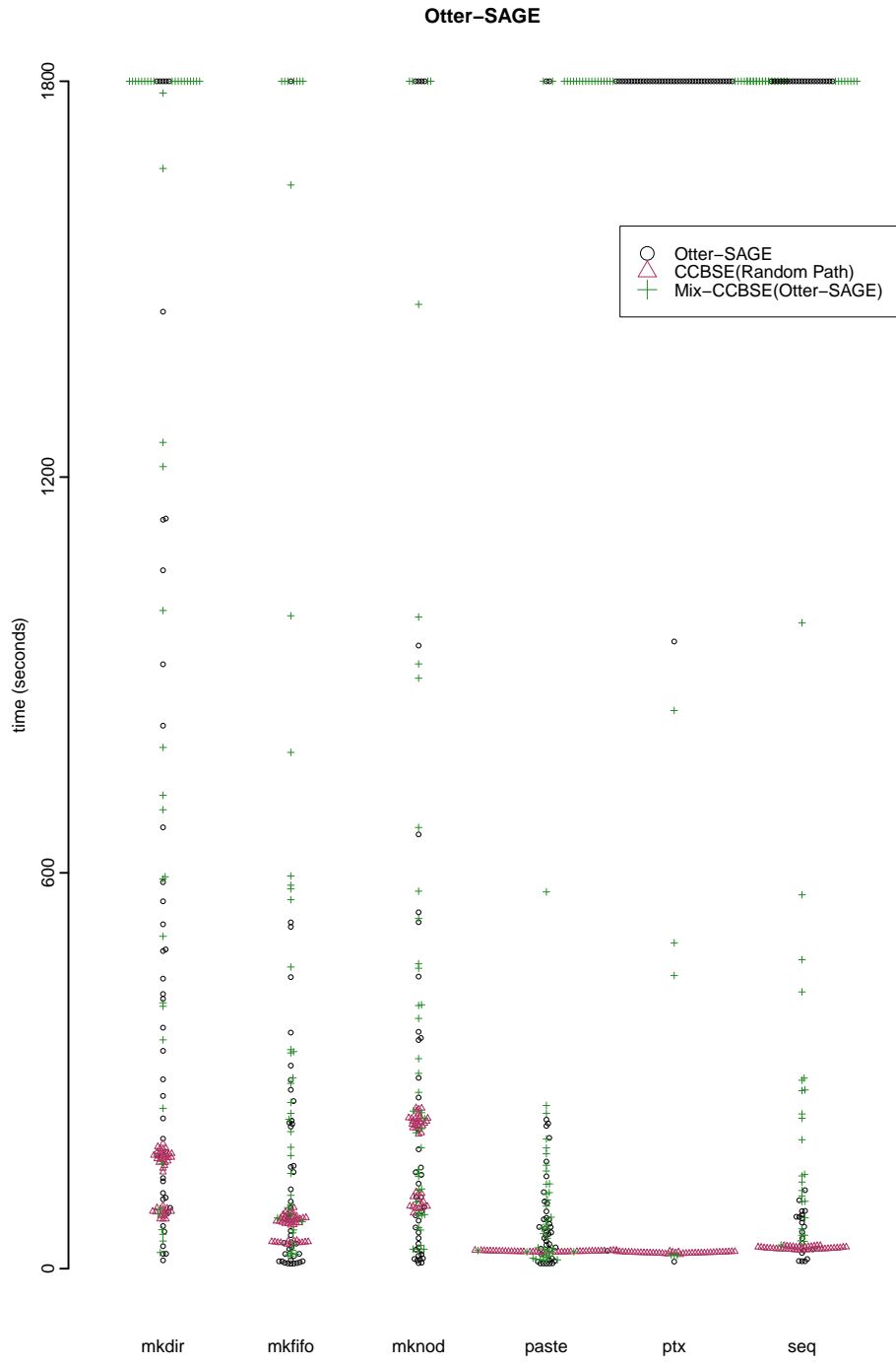**Mix–CCBSE(Otter–SAGE)**

Random Path

Mix−CCBSE(Random Path)

A.2 Overlaid Pure(S), Random Path, Mix-CCBSE(S)

To compare Mix-CCBSE strategies against its components, each of the following plots overlays three beeswarm distribution plots: Pure(S), which is the standard forward strategy S, Random Path, and Mix-CCBSE(S).

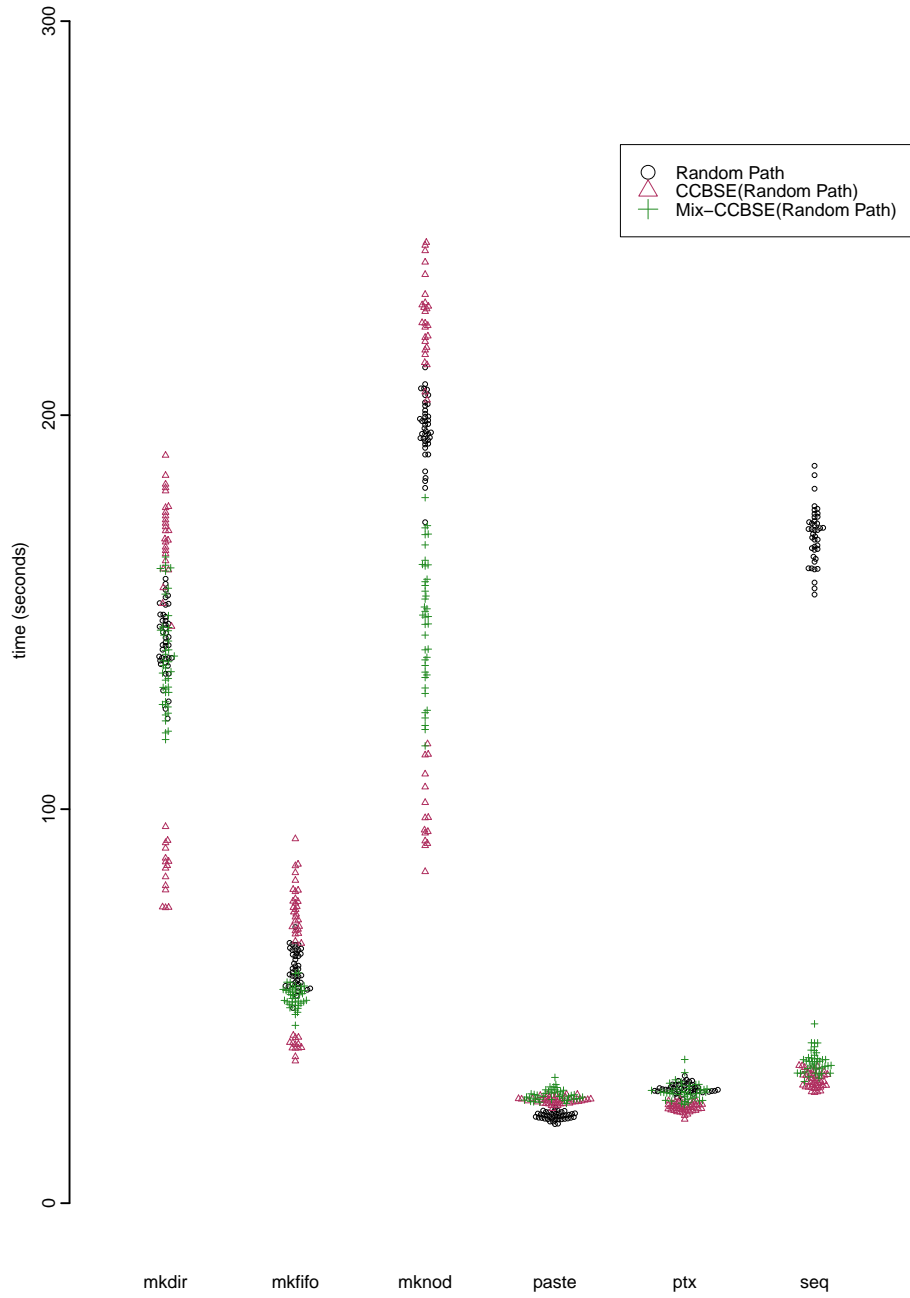# Otter−KLEE

Otter−SAGE

time (seconds)

		Otter−SAGE
		CCBSE(Random Path)
		Mix−CCBSE(Otter−SAGE)

1800

1200

600

0

mkdir   mkfifo   mknod   paste   ptx   seq

Random Path

A.3 Analysis

Many of the distributions in Appendix A.1 are bimodal, which can be seen as two distinct clusters of run times. Since the distributions are observably non-normal, it is inappropriate to summarize our experimental results using mean and standard deviation statistics. Thus, in Table 1, we report the median and SIQR, which are non-parametric (distribution-agnostic) statistics.

Bimodal distribution in CCBSE(RP). CCBSE(RP) is distinctly bimodal for mkdir, mkfifo and mknod, and to a lesser extent for Figure 2. We analyzed these runs and found that, for the faster clusters, CCBSE(RP) found paths from quote to the target line that are also realizable from main. When CCBSE eventually works backwards to main, the search then short-circuits from main through quote to the target line. Thus, these cases demonstrate the advantages of CCBSE.

For the slower clusters, CCBSE(RP) found paths originating from quote that are ultimately not realizable from main. Here, CCBSE(RP) degenerates to pure Random Path with overhead: it works backwards to main (which is the overhead), and then finds a different path to the target. Looking at the Random Path plot in Appendix A.2, we can see that it is indeed the case that the slower cluster in CCBSE(RP) is slightly slower than Random Path.

Bimodal distributions due to time outs. The distributions of several other strategy/program test conditions are also bimodal in that runs either finish quickly or time out. KLEE as well as strategies involving Otter-KLEE and Otter-SAGE seem to exhibit this issue. We speculate that this is due to the coverage-based heuristics used by these strategies: if a run happens to explore paths that cover many of the same lines as the path to the target, the coverage heuristic may then penalize the path to the target, making it more likely to time out. As a result, the timed-out cluster becomes more distinctly separated from the timely clusters, as seen in the plots.

In general, randomness in a strategy can lead to exploration that never reaches the target in certain programs, therefore creating two clusters of timely and timed-out runs.

Mix-CCBSE. At the end of Section 2.3, we explained that we mix strategies with CCBSE in order to get the best of both worlds, but it can as well degenerate to being worse than either. The plots in Appendix A.2 show more examples of the former than the latter.

For Otter-KLEE and Random Path, Mix-CCBSE (as shown by green crosses) tends to be located towards the bottom of the distribution for each program; in the case of Mix-CCBSE(Otter-KLEE) for mknod, it is located at the bottom, i.e., Mix-CCBSE(Otter-KLEE) performs better than either of its constituents alone.

The analysis for Otter-SAGE is less positive: Mix-CCBSE(Otter-SAGE) seems to be as bad as Otter-SAGE alone. We speculate that this is because Otter-SAGE will always run a path to completion, even if the path has reached a point in the

program where the target is no longer reachable, and Mix-CCBSE(Otter-SAGE) can no longer take advantage the partial paths found by CCBSE.