Sparse Symbolic Loop Execution (Registered Report)

Frank Busse
Imperial College London
United Kingdom
f.busse@imperial.ac.uk

Martin Nowack
Imperial College London
United Kingdom
m.nowack@imperial.ac.uk

Cristian Cadar
Imperial College London
United Kingdom
c.cadar@imperial.ac.uk

Abstract

Dynamic symbolic execution is a powerful program analysis technique but is often limited by the path-explosion problem, particularly in the presence of heavily branching loops. In this paper, we introduce *sparse symbolic loop execution* (SSLE), a novel approach aimed at mitigating this issue. SSLE observes the edge patterns of sibling states, spawned from the same loop, at program branches up to a pre-computed loop-impact barrier. States that exhibit unique patterns of taken edges are selected for further exploration, while others are postponed.

We implemented SSLE in a prototype called SparKLE and evaluated it on a set of eight benchmarks against the popular symbolic execution engine KLEE. SSLE shows promising results and could increase line coverage by up to 55% in 1 h runs using a depth-first-search heuristic. In other cases, up to 99.997% of states could be postponed without any loss in coverage. Our planned evaluation will include a diverse set of 50 real-world benchmarks and will aim to better understand the effectiveness of SparKLE in handling symbolic loops, and how it compares with less complex approaches.

CCS Concepts

• **Software and its engineering → Software testing and debugging**.

Keywords

symbolic execution, loops, KLEE

ACM Reference Format:
Frank Busse, Martin Nowack, and Cristian Cadar. 2024. Sparse Symbolic Loop Execution (Registered Report). In *Proceedings of the 3rd ACM International Fuzzing Workshop (FUZZING '24), September 16, 2024, Vienna, Austria.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3678722.3685535

1 Introduction

Dynamic symbolic execution (DSE) [6] is a well-established program analysis technique with applications in many areas, including automatic test generation, bug finding, program repair, reverse engineering, and program verification. It aims to explore all program paths by introducing symbolic input variables, collecting constraints along execution paths, and using a satisfiability modulo theory (SMT) solver to determine path feasibility, check safety properties, and generate concrete inputs for the explored paths.

In practice, the analysis of real-world applications is often limited by two factors: the inherent constraint-solving overhead and *path explosion*. Most applications simply contain too many program paths to explore exhaustively. In the past, several techniques were developed to cope with path explosion, e.g. chopped symbolic execution [23] skips heavily-branching functions and only recovers relevant paths later if necessary, compositional symbolic execution [7] computes and combines function summaries, memoisation [2, 27] stores paths efficiently to disk to recover them later in case the symbolic execution engine runs out of memory, RWSet [1] prunes paths that have an identical suffix with an already explored path, and state merging [12] combines multiple paths into a single one with a more complex (disjunctive) constraint set. Each of these approaches has drawbacks as they rely on trade-offs between path-explosion and constraint-solving complexity, runtime overhead or utilised disk space. For that reason, the most common approach to deal with path explosion is to give up the illusion of being able to cover all paths and solely prioritise interesting ones with sophisticated search heuristics [16].

Significant contributors to path explosion are *heavily-branching loops* with symbolic loop conditions. For instance, when a path with an unconstrained `input` buffer of length `n` reaches a `strlen(input)` function from the C standard library to compute the length of a string in this buffer, a symbolic execution engine would spawn `n` new paths, each having the terminating `0` character in a different position. This means the engine would effectively execute the program under test for all possible lengths of input strings. Evidently, this is in most cases wasted effort and one should restrain the execution to some interesting paths. Interesting candidates for `strlen` are e.g. the empty string, the string lacking `0`-termination, in which case the engine would report an out-of-bounds read, and very few paths in between.

DSE engines come in two flavours: *concolic* engines [8, 18], that are driven by concrete inputs and usually follow single program paths to completion, and *EGT-style* [4, 5] engines, that keep all active paths in memory. We only consider EGT-style engines in this work. Since EGT-style engines explore many execution paths simultaneously and inspect their properties on the fly, the apparent solution to the above problem seems to be to count loop iterations and terminate or postpone paths whose iteration count is considered uninteresting. At a high level, modern greybox fuzzers use a similar approach when tracking edge coverage. Using a bucketing mechanism [29], fuzzers categorise paths with similar edge counts into the same bucket (e.g. 1, 2, 3-4, 5-8, ..., 32-127, 128+), considering them equally interesting. But whereas greybox fuzzers usually rank completed paths and hence can take all coverage information into account, EGT-style DSE engines work with path prefixes and have to look ahead to select promising paths via search heuristics. However, selecting paths just by their iteration count

```
 1 int strcmp(const char *l, const char *r) {
 2   for (; *l==*r && *l; l++, r++);
 3   return *(unsigned char *)l - *(unsigned char *)r;
 4 }
 5
 6 int main(void) {
 7   char input[5];
 8   make_symbolic(input, 5); // make "input" symbolic
 9
10   if (!strcmp(input, "loop")) // decision point
11     puts("loop");
12   if (!strcmp(input, "look")) // decision point
13     puts("look");
14
15   puts("done");
16 }
```

Figure 1: A sample program with a symbolic `input` buffer of length 5, two decision points, and one loop inside `strcmp`.

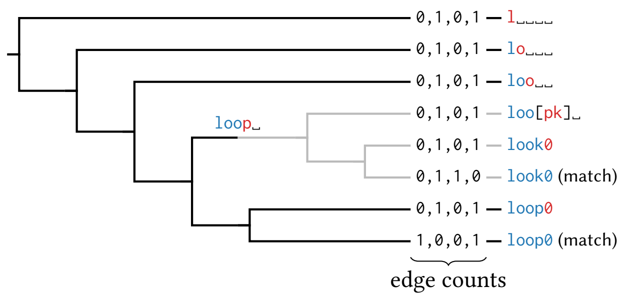

Figure 2: Execution tree for Figure 1. At the end of the execution, each byte in the `input` buffer is either concrete (blue), constrained to exclude certain characters (red) or unconstrained (␣). The edge counts represent the branches taken in the two decision points $\langle L10_t, L10_f, L12_t, L12_f \rangle$.

falls short when later in the control-flow path *decision points* (such as **if** and **switch** statements) depend on specific values altered by a loop. This work aims to mitigate path explosion caused by symbolic loops by prioritising a small subset of paths that effectively exercise all possible behaviours at loop-dependent decision points.

To illustrate the technique, consider the example in Figure 1 where an unconstrained symbolic `input` buffer of length 5 is compared against two concrete strings, 'loop' and 'look'. This code demonstrates a common scenario in which decision points (Lines 10 and 12) depend on values calculated inside a loop (Line 2).

The loop in `strcmp` compares strings character by character, and since the `input` buffer is unconstrained, a DSE engine forks the execution on every comparison. Figure 2 shows all paths and their constraints. The subtree of the second `strcmp` call is shown in grey.

Only a single path that leaves the first `strcmp` is able to branch in the second `strcmp` and match 'look'. After three iterations it has a matching prefix (`loo`) and the constraints on the characters in the fourth (not `p`) and fifth position (unconstrained) permit a match with 'look'. In contrast, a bucketing mechanism that focuses only on certain loop iteration counts (e.g. 0, 1, 2, 4, . . . iterations) will miss this match entirely.

We propose *sparse symbolic loop execution* (SSLE) that aims to find a sweet spot between path explosion and overly aggressive filtering for loop paths, as illustrated by the example above. SSLE distinguishes two types of loops: *simple loops* that do not affect any decision points, and *DP loops* ('DP' for decision point) that affect the outcome of decision points by either modifying variables or adding constraints to symbolic variables.

Paths through symbolic loops classified as *simple loops* are filtered solely by their iteration count: those that do not match 0, 1, 2, 4, 8, 16, . . . , n iterations are postponed.

The handling of *DP loops* is more complex. For all paths that leave a symbolic loop, SSLE counts the edges that are taken at affected decision points, for instance, the *true* and *false* branches at an **if** statement. Then, when a path reaches a pre-computed *loop-impact barrier*, a code location where no relevant decision points can be reached anymore, SSLE compares the taken branches since the path has left the loop with the paths that branched in the same loop but have reached a barrier earlier. If the edge count vector has not been seen before, SSLE keeps exploring that path, otherwise it postpones the exploration of the path. The decision process for postponement can be random or e.g. follow a similar pattern as for the simple loops.

For the program in Figure 1, an impact barrier could be on Line 15 as the **if** statements (decision points) above cannot be reached anymore. The edge count vectors for the individual paths are shown in Figure 2. Only two paths, the matching strings, are able to take one of the *true* branches at the decision points (edge count vectors $\langle 0, 1, 1, 0 \rangle$ and $\langle 1, 0, 0, 1 \rangle$), whereas the six others always take both *false* branches ($\langle 0, 1, 0, 1 \rangle$). In its strictest configuration, SSLE would continue the exploration of the two paths with full matches, select only one of the other paths and hence print 'done' only three times—a path reduction by 62.5% without loss of coverage.

In the remainder of this paper, we describe our approach in §2, discuss our prototype in §3, present preliminary results in §4 and an evaluation plan in §5, before we discuss related work in §6 and conclude in §7.

2 Sparse Symbolic Loop Execution

Sparse symbolic loop execution (SSLE) aims to mitigate path explosion induced by heavily branching symbolic loops. Many of these paths exhibit similar program behaviour and can be deferred or excluded from exploration without diminishing the effectiveness of symbolic execution.

During symbolic execution, an execution path is represented by a *state*, which contains essential runtime information such as a program counter, the call stack, and the set of path constraints. SSLE observes the behaviour of all states leaving a symbolic loop at relevant decision points, e.g. **if** or **switch** statements whose conditions depend on the loop's outcome. States that exhibit similar behaviour by following the same branches are ranked less relevant, postponed from further exploration, and prioritised for termination when a DSE engine runs out of memory.

As already discussed, we distinguish two types of loops: *simple loops*, which do not impact decision points, and *DP loops* that do influence decision points. Since simple loops are merely a special case of DP loops with an empty set of decision points, we explain the handling of DP loops first in §2.1 and then simple loops in §2.2.

2.1 Loops Affecting Decision Points (DP loops)

As discussed in §1, states should not simply be postponed based on their iteration count when the computation inside a loop affects the outcome of decision points. Edges in the follow-up control flow could be missed, resulting in uncovered code sections. To still be able to postpone a significant amount of states, SSLE observes their behaviour at relevant decision points up to specific loop-impact barriers and ranks them according to the uniqueness of their behaviour.

SSLE does this by tracking the traversed decision point edges for an execution state until it hits a loop-impact barrier. As the barrier is reached, the state is categorised using the edge behaviour, with only a few witness states per category selected for further exploration and others postponed.

In the following, we will discuss our approach in more detail using the example shown in Figure 3. The program computes the length (Line 23) of a symbolic string buffer of maximum length 10. Depending on the length of the input, the program will either output `odd`, `even`, `2` or `empty`.

During symbolic execution, the initial program state reaches the loop inside `strlen` (Line 13), which compares each character in the input buffer against the `0`-termination character. Since the `input` buffer is unconstrained, the DSE engine spawns ten new states, each representing a different number of iterations through the loop and thereby different lengths of the `input` string. The state representing the last iteration lacks the `0`-termination character, causes an out-of-bounds read, which the DSE engine reports, and gets terminated immediately. For all other states, and depending on the returned length (`len`), the function pointer `print` gets initialised with different printing functions (Lines 28, 30 and 32), and called (Line 34), before the program terminates with the message 'done' (Line 35). The printing functions output 'empty', 'odd' and 'even' (Lines 1, 2 and 8) unless the length equals 2, in which case `print_even` outputs '2' (Line 6).

```
1  void print_empty(size_t l) { puts("empty"); }
2  void print_odd(size_t l) { puts("odd"); }
3
4  void print_even(size_t l) {
5    if (l == 2) // DP 3
6      puts("2"); // barrier 2
7    else
8      puts("even"); // barrier 3
9  }
10
11 size_t strlen(const char *s) {
12   const char *a = s;
13   for (; *s; ++s); // loop exit triggers analysis
14   size_t result = s - a;
15   return result;
16 }
17
18 int main(void) {
19   void (*print)(size_t) = NULL;
20   char input[10];
21   make_symbolic(input, 10); // symbolic input
22
23   size_t len = strlen(input); // [0-9] + OOB
24
25   if (len) { // DP 1
26     bool is_odd = len % 2;
27     if (is_odd) // DP 2
28       print = print_odd;
29     else
30       print = print_even;
31   } else
32     print = print_empty;
33
34   print(len);
35   puts("done"); // barrier 1
36 }
```

Figure 3: A program with a symbolic `input` buffer of length 10. The state entering the `strlen` function spawns ten children for different string lengths and one out-of-bounds read for the last iteration lacking the terminating `0` character.

2.1.1 *Identifying Decision Points.*

Since `strlen` is a state-spawning *symbolic loop*, SSLE first identifies decision points by tainting all values and memory locations that might be affected by that loop. In our example, the **`if`** statements in Lines 5, 25 and 27 would be identified as decision points since `len`, `is_odd` and `l` will get tainted starting from the loop in `strlen`.

The tainting process is initiated when a state leaves a specific symbolic loop for the first time. It starts with the loop and taints all values and memory objects within the loop. The process continues to recursively taint all values and memory objects that are affected by tainted values in the function containing the loop. Only code paths that either reach or are reachable from the loop are considered. Notably, the algorithm considers only data dependencies and not control dependencies.

It then proceeds to inspect the *call stack* of the current state and walks up the stack frames along the call sites continuing the tainting process. However, tainting only along the call stack is often not sufficient. For instance, in our example, SSLE would not observe the special behaviour for '2' in `print_even`. To address this issue, SSLE also continues tainting within called functions. The tainting continues until it reaches a fixed-point set of tainted values. Although computationally expensive, computing a fixed-point set enables tainted values to propagate back into their calling functions and vice versa, enabling SSLE to identify more decision points and differentiate more loop states.

Table 1: The edge counts for Figure 3. Only four states (greyed out) have unique edge count vectors. The out-of-bounds (OOB) state gets terminated immediately.

State (by string length)	Edge counts					
	$DP1_t$	$DP1_f$	$DP2_t$	$DP2_f$	$DP3_t$	$DP3_f$
0	0	1	0	0	-	-
1	1	0	1	0	-	-
2	1	0	0	1	1	0
3	1	0	1	0	-	-
4	1	0	0	1	0	1
5	1	0	1	0	-	-
6	1	0	0	1	0	1
7	1	0	1	0	-	-
8	1	0	0	1	0	1
9	1	0	1	0	-	-
OOB	0	0	0	0	-	-

SSLE utilises configurable thresholds for both the stack traversal (*callsite depth*) and the call chain traversal (*call depth*) during tainting. An unrestricted analysis would not only significantly increase the analysis time but also detect more affected decision points and, hence, barriers further away from the loop, thus delaying the postponement of states. With thresholds, we aim to find a sweet spot between postponing too many states (and losing coverage) and path explosion.

2.1.2 Identifying Loop-impact Barriers. Next, SSLE computes the set of *loop-impact barriers*. Barriers are the first instructions along a code path after a (call to a) decision point that cannot reach another (call to a) decision point. For the program outlined in Figure 3, three barriers are computed. If `print` points to `print_even`, the barriers are located in Lines 6 and 8. Otherwise, the barrier is in Line 35. Our current implementation's barrier computation is less precise in this case and assumes that the `print` call could still reach decision point 3 (Line 5) via a function pointer (`print_even`). A more precise, context-sensitive implementation would place two barriers in Lines 28 and 32 instead.

2.1.3 Edge Tracking. The core idea of sparse symbolic loop execution is to categorise states that branched in the same loop via their behaviour at decision points and select only a few representatives from each category. SSLE observes their behaviour by counting how often a state traverses the edges of affected decision points. In our example, there are only four different edge count patterns (or vectors) for all states as shown in Table 1. For instance, state 0, representing the empty string, is the only state that takes the false branch ($DP1_f$) of the first decision point (Line 31) and is therefore considered an interesting state.

2.1.4 Postponing States. SSLE is search heuristic-agnostic and does not influence the state selection process. However, whenever a state reaches a loop-impact barrier, its edge count pattern is compared against those of its forked states from the same loop entry that have previously reached a loop-impact barrier. If its edge count pattern is unique, the state is retained; otherwise, it may be postponed.

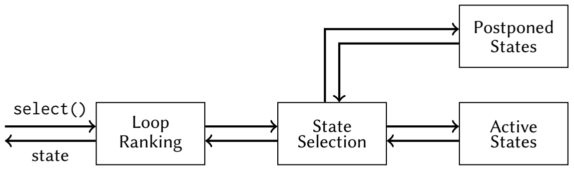

Figure 4: SSLE's state selection strategy.

Various strategies can be applied when postponing states. The strictest strategy would keep only one representative per edge count pattern occurrence, while other strategies could select randomly from a pattern group or, for example, keep the 0, 1, 2, 4, 8, and so on states per group.

Figure 4 shows the state selection strategy of SSLE. The *loop ranking* module is responsible for the different postponement strategies (e.g. keep only one witness state for each edge count vector) and marks states for postponement. In contrast, the *state selection* module implements the actual state selection. All states are divided into a pool of active states and a pool of postponed states. The *state selection* module chooses active states with a high probability and postponed states with a low probability (in both cases according to the underlying search strategy used by the symbolic execution engine).

One possible strategy is to set the probability of selecting postponed states to zero, i.e. to discard postponed states. Such a strategy can be useful to reduce memory pressure. DSE engines often suffer from memory exhaustion due to the significant number of states generated and may resort to randomly selecting states for early termination [2]. Thus, by terminating postponed states immediately, SSLE can reduce memory pressure.

2.2 Simple Loops

Simple loops are a special case of DP loops with an empty set of relevant decision points. They are rare in practice and can often be attributed to imprecise taint analysis. Since there are no decision points to observe, they are solely ranked by their number of iterations in a loop. Once more, several filter strategies could be implemented, from random selection to employing a bucketing scheme similar to modern greybox fuzzers.

3 Implementation

We have implemented sparse symbolic loop execution (SSLE) on top of KLEE [4], a state-of-the-art dynamic symbolic execution engine, and named our prototype SparKLE. SparKLE operates on LLVM [13] bitcode and employs SVF's [21] Andersen-Wave points-to analysis for the tainting stage.

We tried to interfere as little as possible with KLEE's core components and mainly added more statistics and hooks to its execution engine to call our *loop handler*. The loop handler monitors loop entries and exits, initiates tainting, counts iterations and tracks edges at reached decision points. Besides that, we added the modules of Figure 4 to KLEE's chain of search heuristics.

In the following sections, we describe some implementation details regarding state batching (§3.1), tainting (§3.2), DP loops (§3.3) and simple loops (§3.4).

3.1 State Batching

KLEE provides a *batching searcher*, which sits in front of the actual searcher and prioritises the same state for a fixed number of instructions or a given amount of time before the actual searcher is called. Its goal is to reduce the computational overhead of some search heuristics if called for every executed instruction as they require tree traversals or distance computations.

We extended the default batching to additionally query the underlying search heuristic when the currently explored state just branched, left a simple loop, reached a barrier or got postponed.

3.2 Tainting

When SparKLE counts the depth of call sites or call chains to limit tainting, it does not count 'trivial' functions that consist of only a single basic block. Based on our initial experiments, some tested applications use this typical software pattern to provide a public API interface that simply forwards to internal functions for encapsulation.

A decision we faced during development was whether to increase the precision of SVF's points-to analysis by integrating it with KLEE. At runtime, KLEE collects constraints along paths, including those related to memory objects, which would allow us to reduce the points-to sets of SVF's analysis to the values that are feasible along the current path (similar in spirit to past-sensitive pointer analysis [22]). Fewer tainted values can result in fewer affected decision points and, therefore, barriers closer to loops. However, since we use memoisation heavily, we would need to recompute the analysis when constraints change. We decided against it and in favour of a simpler implementation, but this trade-off needs to be evaluated in the future.

3.3 DP loops

A noteworthy implementation detail where SparKLE differs from SSLE is that it does *not* store edge count vectors for every state but only uses cumulative hashes of tracked edges. This reduces the memory footprint of SparKLE but has the disadvantage that the edge tracking in comparison to SSLE is order-dependent.[1] There might be cases, especially with nested loops, where this approach distinguishes more states than necessary.

3.4 Simple Loops

Simple loops are postponed solely by their iteration count. In its default mode, SparKLE keeps states with iteration counts of 0, 1, 2, 8, … and the last iteration and postpones all others.

Keeping the state with the highest iteration count in dynamic symbolic execution is non-trivial. States might spawn new children due to conditional branch instructions inside the loop body such that states with different or even the same iteration counts can exist at any time. Since search heuristics can select arbitrary states and SparKLE cannot predict what state would create the deepest execution subtree, we implemented the ranking of the 'last' iteration on a best-effort basis: only when there is no state in the loop with the same iteration count and no other state associated with that loop entry had a higher or equal iteration count before, the state is kept and otherwise postponed.

[1] `hash(A, hash(B))` $\neq$ `hash(B, hash(A))` whereas the edge count vector $\langle 1, 1\rangle_{A,B}$ remains identical for two edges A and B, regardless of the order in which they are traversed.

Table 2: Loop statistics for several benchmarks (1 h DFS, 3/3).

App	Total	Reached	Symbolic	Branches
basename	113	56	7	1,550,793
cat	121	59	7	282
cxxfilt	218	61	14	106,705
date	241	100	33	11,144
factor	244	58	16	125
raw2tiff	450	77	12	348,072
sum	122	73	11	13,811
uniq	122	68	20	330,262

Table 3: Covered bitcode lines for different thresholds (1 h DFS). Best-performing configurations are shown in bold.

		Tainting thresholds		
App	Baseline	2/2	3/3	4/4
basename	**5,285**	**5,285**	**5,285**	**5,285**
cat	**5,767**	**5,767**	**5,767**	**5,767**
cxxfilt	6,481	**6,769**	6,481	6,481
date	10,454	**11,656**	10,901	10,888
factor	4,154	**6,440**	4,169	4,154
raw2tiff	9,398	**9,405**	9,354	9,347
sum	7,522	**7,872**	**7,872**	7,522
uniq	7,676	8,280	**8,286**	8,196

Evidently, a similar issue occurs when one wants to keep the second to last iteration, which is currently not supported by SparKLE. However, it could be helpful in cases where the last iteration leads to an out-of-bounds access.

4 Preliminary Results

For experiments with our early SparKLE prototype, we have selected eight benchmarks: *cxxfilt* from GNU Binutils, *raw2tiff* from LibTIFF, and *basename*, *cat*, *date*, *factor*, *sum* and *uniq* from GNU Coreutils.

We run each benchmark for 1 h, using the depth-first search (DFS) heuristic. SparKLE's tainting was limited to three call sites along the stack and three calls down from every stack level. (We denote these thresholds using the notation call stack depth / call depth, in this case 3/3.)

Table 2 presents loop statistics for these runs that show that loops play a significant role in symbolic execution. The bitcode files[2] contain between 113 and 450 *total* loops, and between 56 and 100 are *reached*. In the case of *basename*, as few as seven loops spawn new states (*symbolic*), but this is enough to create as many as 1,550,793 new states in loop basic blocks (*branches*).

Table 3 presents the results for different thresholds, showing that SparKLE is effective and outperforms the baseline. The 2/2

[2] KLEE operates at the level of LLVM bitcode [13].

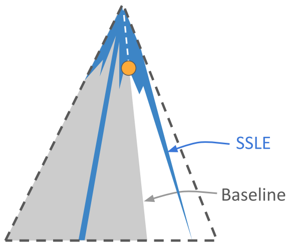

Figure 5: Baseline progresses to explore the execution tree with DFS while SSLE overtakes it at the highlighted node (●).

Table 4: Analysis times in seconds for 1 h DFS runs (rounded to whole seconds).

		Tainting thresholds		
App	SVF	2/2	3/3	4/4
basename	0	0	2	6
cat	0	1	3	12
cxxfilt	1	2	15	235
date	1	29	357	2,572
factor	1	1	4	18
raw2tiff	1	2	7	22
sum	0	1	5	12
uniq	0	3	32	174

threshold obtains the best results overall, achieving the highest coverage in all cases except for *uniq*, where it loses by only a few lines. In some cases, such as for *factor* and *date*, the gain is significant.

However, although the 2/2 threshold shows the best results coverage-wise, it lacks *precision*, which means it postponed paths and consequently missed code sections that were covered with the baseline. A straightforward method to measure precision for DFS runs is to compare the execution trees of SparKLE with those of the baseline execution. For this purpose, SparKLE annotates execution trees with timestamps and coverage information, and stores them to disk. Due to heavy pruning, SparKLE should be able to 'overtake' the baseline execution at a specific node and create a wider, but shallower, tree. Figure 5 illustrates graphically how the exploration space of SparKLE compares to the baseline. SparKLE prunes a large part of the execution tree, and from the highlighted node starts to overtake the DFS exploration of the baseline. If the pruning is precise, no coverage is lost, but thresholds can introduce imprecision.

For instance, with 2/2 thresholds, SparKLE *overtakes* the 1 h baseline run of *cxxfilt* in less than a second by pruning large subtrees but has only covered 2720 lines at that point. The tainting is therefore imprecise and misses code lines that would be coverable along such paths. However, in the remaining time, it covers all missed lines plus 288 additional ones, a 4.4% increase. The 3/3 and 4/4 thresholds create too much overhead and too long distances to loop-impact barriers in this case: 3/3 overtakes baseline with the current prototype only 1 min before the 1 h timeout whereas 4/4 is several minutes behind.

Table 5: Ratio of terminated states in percent.

	Simple			DP-loops		
App	2/2	3/3	4/4	2/2	3/3	4/4
basename	0	0	0	99.997	99.997	99.996
cat	0	0	0	48.465	48.330	48.330
cxxfilt	0	0	0	99.948	99.942	99.935
date	0	0	0	97.885	96.696	95.703
factor	0	0	0	38.130	16.822	17.143
raw2tiff	0	0	0	99.923	99.915	99.838
sum	0.387	0.301	0	98.524	54.294	0.079
uniq	4.661	12.499	0.537	94.378	85.907	99.364

Table 4 shows the analysis times of SparKLE for different thresholds. With the exception of *date*, the analysis times are small relative to the 1 h time budget. *date* is an outlier that we need to investigate further—but even for threshold 4/4 where SparKLE spends most of its run time tainting (43 min), it still manages to cover more lines than baseline KLEE.

All experiments above use the strictest filtering mechanism for DP loops that retains only a single state per edge pattern. Table 5 shows the ratio of terminated states for different thresholds. The number of terminated states is significant; however, with high thresholds, the barriers are closer to the program exit of a path and SSLE is less effective.

Simple loops are rare, and the decrease in filtered states with higher thresholds suggests that such loops only occur due to imprecise tainting. As a result, decision points that are not present in the direct parent functions are therefore missed, causing the respective loops to be treated as simple loops.

5 Planned Evaluation

In the following, we describe the experiments we intend to conduct to evaluate the efficacy of SSLE. We first outline the research questions we aim to answer in §5.1, then describe the benchmark set in §5.2 and finally explain the proposed experiments and evaluation metrics in §5.3.

5.1 Research Questions

The evaluation aims to address two key research questions:

RQ1 Is SSLE an effective approach to postpone or filter states, thereby reducing path explosion?

RQ2 How does SSLE compare to less complex approaches? We propose two alternative approaches: a) treating all loops as *simple loops* and b) postponing states based on a fixed number of decision points without tainting.

The comparison with alternative approaches in the second research question is supposed to address the fundamental question: whether SSLE's complexity is truly necessary or if similar results can be achieved with less complex approaches.

5.2 Benchmarks

We plan to select a diverse set of 50 real-world benchmarks from different tool suites and libraries, such as *GNU Awk*, *GNU bc*, *GNU*

Table 6: RQ1 experiment sets.

	Tainting thresholds	DP filter	Revival
I	1/1 2/0 2/1 2/2 3/1 3/2 3/3 4/2 4/3	one	0%
II	1/1 2/0 2/1 2/2 3/1 3/2 3/3 4/2 4/3	binary	0%
III	1/1 2/0 2/1 2/2	one	10%
IV	1/1 2/0 2/1 2/2	one	20%

Binutils, *GNU Coreutils*, *GNU Grep*, *libsndfile*, *LibTIFF* and *libxml*. To understand the impact of loops on these applications, we are going to report several statistics of SparKLE:

- detected loops in a bitcode file,
- covered loops,
- reached pre-headers across all paths,
- reached loops containing symbolic branches and
- branches in loop basic blocks

5.3 Experiments

5.3.1 RQ1: Efficacy. To evaluate SSLE's efficacy, we need to evaluate various combinations of SparKLE's configuration options. The three most important configuration options are:

(1) Tainting thresholds (call stack depth / call depth): Higher thresholds result in a more precise decision point and barrier computation, but also move barriers further away from the respective loop. Consequently, more code needs to be executed before a state gets postponed. However, this makes it less likely that uncovered code is missed due to an early cut-off of a subtree in the execution tree.
(2) State revival probability: The possibility of reviving postponed states (e.g. every 10th state gets selected from the postponed set).
(3) Filter strategies: how many and what states are selected for each DP pattern (e.g. only one, every 1, 2, 8, ...).

Table 6 shows the different SparKLE configurations that we plan to run. All configurations, plus KLEE as a baseline, will be run for 1 h with the DFS search heuristic. DFS is highly deterministic and tends to fully explore loops further down the execution paths.

Experiment sets I and II shown in Table 6 use different thresholds for the tainting phase and compare two filtering strategies for DP loops: selecting *one* witness state per DP pattern, and a less aggressive version that selects every 1, 2, 8, 16, 32, ... state reaching a barrier with a specific edge pattern (*binary*).

Experiment sets III and IV assume that lower thresholds with a more aggressive filtering strategy miss uncovered lines (in comparison to baseline KLEE) due to heavy path pruning. In these experiments, we aim to evaluate if state revival can mitigate this issue by selecting 10% and 20% of states from the postponed set, respectively.

We use the line coverage information provided by KLEE as our primary metric. Of course, the higher the coverage the better. However, even with higher coverage, SSLE might miss lines that are covered by baseline KLEE. To make a comparison more meaningful, we additionally store the execution trees of all runs on disk and compare them with baseline KLEE. With DFS, it is straightforward to determine a node where one run *overtakes* another, as discussed in §4. At this node, we will compare timestamps and coverage information to assess the precision of SSLE. If the coverage is lower at that node, SSLE has been imprecise. If the coverage is the same but the node was reached much earlier, SSLE was effective. We are aware that changes in the environment can lead to divergences [2] that would render tree comparisons meaningless, for instance, because a subtree got removed due to memory exhaustion rather than SSLE. If such divergences affect too many benchmarks, we report precision based on the set differences of covered lines between baseline and SparKLE.

Due to its exhaustiveness, DFS is not the best search heuristic for quickly covering large portions of code. Therefore, KLEE alternates between a *random-path* traversal and a *distance-based* (to uncovered code) selection of states in its default mode. While effective, this heuristic has several shortcomings when used as a baseline:

(1) It is highly non-deterministic when used correctly with different input seeds for KLEE's random-number generator across runs.
(2) It creates wide execution trees with many dangling states. This leads to memory pressure and early state termination, implicitly pruning subtrees randomly.
(3) It prioritises shallow paths over deeper paths, meaning that loops are typically not exhaustively explored.

To obtain statistically robust results for this heuristic, we plan to run 10 randomly selected benchmarks 100 times each, for one hour, using different seeds for KLEE's random number generator. This will be conducted across three configurations: baseline KLEE, and SparKLE utilising the two most successful configurations in terms of coverage identified from the DFS experiments. We will compare the combined coverage of each configuration to determine if our SSLE prototype, SparKLE, can cover more lines of code or at least different code sections that were inaccessible before.

5.3.2 RQ2: Other Approaches. Tainting and bookkeeping in SSLE can be costly, especially with high tainting thresholds. We propose two less complex approaches and compare their efficacy against SSLE with 1 h DFS runs:

all-simple treats all loops as simple loops without decision points and selects states solely by their iteration count.

fixed-DP skips the tainting step and tracks edge patterns only for a fixed number of DPs.

We plan to implement these approaches in SparKLE and compare those versions based on the same benchmark set for 1 h with DFS search heuristic.

For the *all-simple* approach, we already implemented a prototype and the preliminary results are shown in Table 7. This approach outperforms SSLE (with the best-performing 2/2 configuration) for *cat*, and baseline KLEE for three more benchmarks. These results suggest that while the all-simple approach can be effective for certain benchmarks, it generally falls short compared to the more sophisticated SSLE method.

Although less precise, the *fixed-DP* approach has less computational overhead due to the missing tainting step and serves as an indicator of whether SSLE's complexity is actually needed to reduce path explosion with reasonable precision. We propose 8, 16,

Table 7: Covered bitcode lines for the *all-simple* approach (1 h DFS, no state revival, simple filter: binary) in comparison to baseline KLEE and SSLE with 2/2 thresholds (Table 3).

App	Baseline	All-simple	SSLE 2/2
basename	5,285	5,285	5,285
cat	5,767	**6,437**	5,767
cxxfilt	6,481	3,516	**6,769**
date	10,454	10,641	**11,656**
factor	4,154	4,178	**6,440**
raw2tiff	9,398	9,247	**9,405**
sum	7,522	7,522	**7,872**
uniq	7,676	7,887	**8,280**

32, 48, and 64 as static limits for the number of tracked decision points and will use higher numbers if initial results indicate that it might be worthwhile.

After completing the implementation, we will provide an artefact containing SparKLE, all benchmarks and our results.

6 Related Work

Path explosion caused by loops is a well-known problem and several mitigations [25] have been proposed in the past. The simplest approach is *bounded symbolic execution* that just limits the number of iterations to a fixed number and rarely fully explores interesting loops. Readily available are *compiler optimisations*, which work well in practice for some types of loops [3, 24]. A more labour-intensive approach involves writing *models* for the most common loops in functions such as `strlen`, `strcmp` or `memcmp`. Depending on the implementation, these models can reduce the number of paths significantly, but in the case of complete models, they can also increase the constraint-solving overhead due to more complex path constraints. To handle operations on strings, typically involving heavily-branching loops, specialised *string solvers* [11, 30] can be employed to solve more abstract constraints in the theory of strings. However, to be most effective they rely on well-defined string-APIs and require e.g. indices into strings to be integers. For C-like languages, strings are often manipulated via hand-written loops [10] and, in the case of KLEE, indices are just bitvectors.

Instead of unrolling loops during execution, *loop summarisation* [9, 17, 26] tries to create summaries for loops by introducing symbolic iteration counts and capturing a loop's behaviour in more complex path constraints. This technique is successfully used in practice, but is not applicable to all loops. A similar approach is *compact symbolic execution* [20] that uses parametric templates to describe all states leaving cyclic program paths. This can speed up the execution and reduce the number of states but also complicates constraint solving as it introduces path constraints with quantifiers.

Sparse symbolic loop execution (SSLE), as proposed in this paper, is orthogonal to most approaches or can be applied when other approaches fail. The proposed sampling of intermediate loop iterations resembles the bucketing algorithm of common AFL-based fuzzers [28], although in a look-ahead manner. SSLE primarily acts as a filter for loop states and is search heuristic-agnostic. However, the outcome of branch decisions has been used to steer the exploration of symbolic execution engines before.

Context-guided search (CGS) [19] explores programs incrementally in a breadth-first-search manner and selects paths for further exploration by comparing their (increasing) *k-contexts*. A *k-context* is the suffix sequence of taken branches of length *k* of a state. Paths that have an unseen *k-context* are prioritised. Li et al. [15] propose *subpath-guided search* (SGS) as a method to direct the search process along less-explored paths. The approach utilises a priority queue where states are ordered based on the frequency of their recent branch decisions, specifically the last *n* decisions, referred to as a *subpath*. States that follow less frequently observed subpaths are prioritised, thereby encouraging exploration in under-explored regions of the state space.

Greybox fuzzing faces the same challenges as symbolic execution in the context of path explosion and similar mitigation techniques can be applied. Instead of using sliding context windows or path suffixes, *prefix-guided* fuzzing [14] compares prefixes of traversed basic block edges and terminates executions with known prefixes early. The prefix length is determined for each benchmark individually by sampling known inputs and finding a setting with a high recall rate. In contrast, SSLE does not consider global prefixes across all program paths but instead focuses on statically computed prefixes among paths originating from the same loop. This approach could be beneficial for greybox fuzzing, especially for paths that branch deeply within the code, making it an interesting direction for future research.

7 Conclusion

In this paper, we have introduced *sparse symbolic execution* (SSLE), a novel approach designed to mitigate the path-explosion problem caused by heavily branching loops during dynamic symbolic execution. After classifying sibling states spawned from the same loop according to their observed behaviour at relevant decision points, SSLE selects only a small number of states from each class and postpones others. We have implemented SSLE in a prototype based on KLEE, called SparKLE, and demonstrated its efficacy for a small number of benchmarks as preliminary results.

We have also outlined a plan for a more comprehensive evaluation where we will evaluate SparKLE with different configurations against a large set of benchmarks and also less complex alternative approaches.

8 Acknowledgements

This project has received funding from the European Research Council under the European Union's Horizon 2020 research and innovation program (grant agreement 819141) and from the Engineering and Physical Sciences Research Council (grant EP/X040836/1).

References

[1] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. 2008. RWset: Attacking Path Explosion in Constraint-Based Test Generation. In *Proc. of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)* (Budapest, Hungary).

[2] Frank Busse, Martin Nowack, and Cristian Cadar. 2020. Running Symbolic Execution Forever. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'20)* (Online). https://doi.org/10.1145/3395363.3397360

[3] Cristian Cadar. 2015. Targeted program transformations for symbolic execution. In *Proc. of the Joint Meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering, New Ideas Track (ESEC/FSE NI'15)* (Bergamo, Italy). https://doi.org/10.1145/2786805.2803205

[4] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)* (San Diego, CA, USA).

[5] Cristian Cadar and Dawson Engler. 2005. Execution Generated Test Cases: How to Make Systems Code Crash Itself. In *Proc. of the 12th International SPIN Workshop on Model Checking of Software (SPIN'05)* (San Francisco, CA, USA). https://doi.org/10.1007/11537328_2

[6] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Communications of the Association for Computing Machinery (CACM)* 56, 2 (2013), 82–90. https://doi.org/10.1145/2408776.2408795

[7] Patrice Godefroid. 2007. Compositional Dynamic Test Generation. In *Proc. of the 34th ACM Symposium on the Principles of Programming Languages (POPL'07)* (Nice, France).

[8] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'05)* (Chicago, IL, USA).

[9] Patrice Godefroid and Daniel Luchaup. 2011. Automatic Partial Loop Summarization in Dynamic Test Generation. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'11)* (Toronto, Canada).

[10] Timotej Kapus, Oren Ish-Shalom, Shachar Itzhaky, Noam Rinetzky, and Cristian Cadar. 2019. Computing Summaries of String Loops in C for Better Testing and Refactoring. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'19)* (Phoenix, AZ, USA).

[11] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. 2009. HAMPI: A Solver for String Constraints. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'09)* (Chicago, IL, USA).

[12] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient state merging in symbolic execution. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'12)* (Beijing, China).

[13] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. of the 2nd International Symposium on Code Generation and Optimization (CGO'04)* (Palo Alto, CA, USA). https://doi.org/10.1109/CGO.2004.1281665

[14] Shaohua Li and Zhendong Su. 2023. Accelerating Fuzzing through Prefix-Guided Execution. In *Proc. of the ACM on Programming Languages (OOPSLA'23)* (Cascais, Portugal).

[15] You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. 2013. Steering Symbolic Execution to Less Traveled Paths. In *Proc. of the 28th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'13)* (Indianapolis, IN, USA). https://doi.org/10.1145/2509136.2509553

[16] Arash Sabbaghi and Mohammad Reza Keyvanpour. 2020. A Systematic Review of Search Strategies in Dynamic Symbolic Execution. *Computer Standards & Interfaces* 72 (2020). https://doi.org/10.1016/j.csi.2020.103444

[17] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. 2009. Loop-extended Symbolic Execution on Binary Programs. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'09)* (Chicago, IL, USA).

[18] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proc. of the Joint Meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'05)* (Lisbon, Portugal). https://doi.org/10.1145/1081706.1081750

[19] Hyunmin Seo and Sunghun Kim. 2014. How We Get There: A Context-Guided Search Strategy in Concolic Testing. In *Proc. of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'14)* (Hong Kong). https://doi.org/10.1145/2635868.2635872

[20] Jiří Slabý, Jan Strejček, and Marek Trtík. 2013. Compact Symbolic Execution. In *Automated Technology for Verification and Analysis (ATVA)*.

[21] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural static value-flow analysis in LLVM. In *Proc. of the 25th International Conference on Compiler Construction (CC'16)* (Barcelona, Spain).

[22] David Trabish, Timotej Kapus, Noam Rinetzky, and Cristian Cadar. 2020. Past-Sensitive Pointer Analysis for Symbolic Execution. In *Proc. of the Joint Meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'20)* (Online).

[23] David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. 2018. Chopped Symbolic Execution. In *Proc. of the 40th International Conference on Software Engineering (ICSE'18)* (Gothenburg, Sweden).

[24] Jonas Wagner, Volodymyr Kuznetsov, and George Candea. 2012. -Overify: Optimizing Programs for Fast Verification. In *Proc. of the 14th Workshop on Hot Topics in Operating Systems (HotOS'13)* (Santa Ana Pueblo, NM, USA).

[25] Xusheng Xiao, Sihan Li, Tao Xie, and Nikolai Tillmann. 2013. Characteristic Studies of Loop Problems for Structural Test Generation via Symbolic Execution. In *Proc. of the 28th IEEE International Conference on Automated Software Engineering (ASE'13)* (Palo Alto, CA, USA). https://doi.org/10.1109/ASE.2013.6693084

[26] Xiaofei Xie, Yang Liu, Wei Le, Xiaohong Li, and Hongxu Chen. 2015. S-looper: Automatic Summarization for Multipath String Loops. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'15)* (Baltimore, MD, USA).

[27] Guowei Yang, Corina S. Păsăreanu, and Sarfraz Khurshid. 2012. Memoized symbolic execution. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'12)* (Minneapolis, MN, USA).

[28] Michal Zalewski. [n. d.]. Technical "whitepaper" for afl-fuzz. http://lcamtuf.coredump.cx/afl/technical_details.txt.

[29] Michał Zalewski. 2019. AFL buckets. https://afl-1.readthedocs.io/en/latest/about_afl.html#detecting-new-behaviors.

[30] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. 2013. Z3-str: A Z3-based String Solver for Web Application Analysis. In *Proc. of the Joint Meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'13)* (Saint Petersburg, Russia).

Received 2024-06-21; accepted 2024-07-22