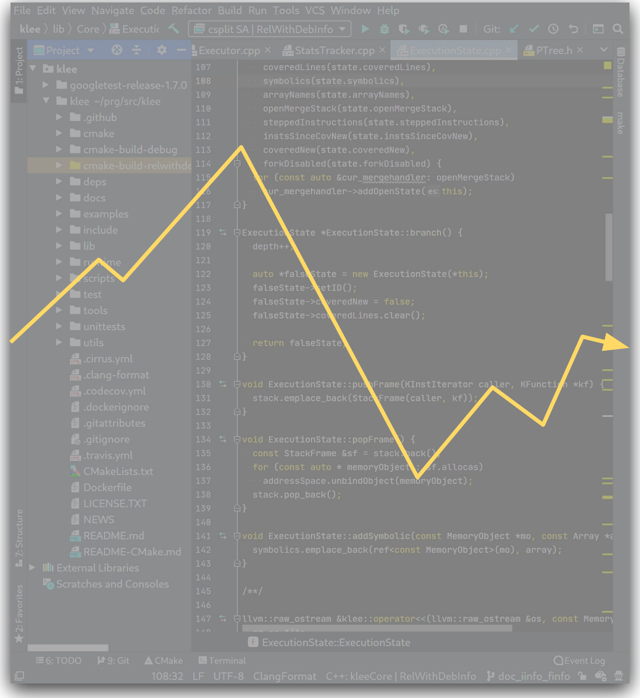
Running Symbolic Execution Forever

Frank Busse · Martin Nowack · Cristian Cadar
Imperial College London

ISSTA 2020, 18-22 July, Virtual Conference, USA

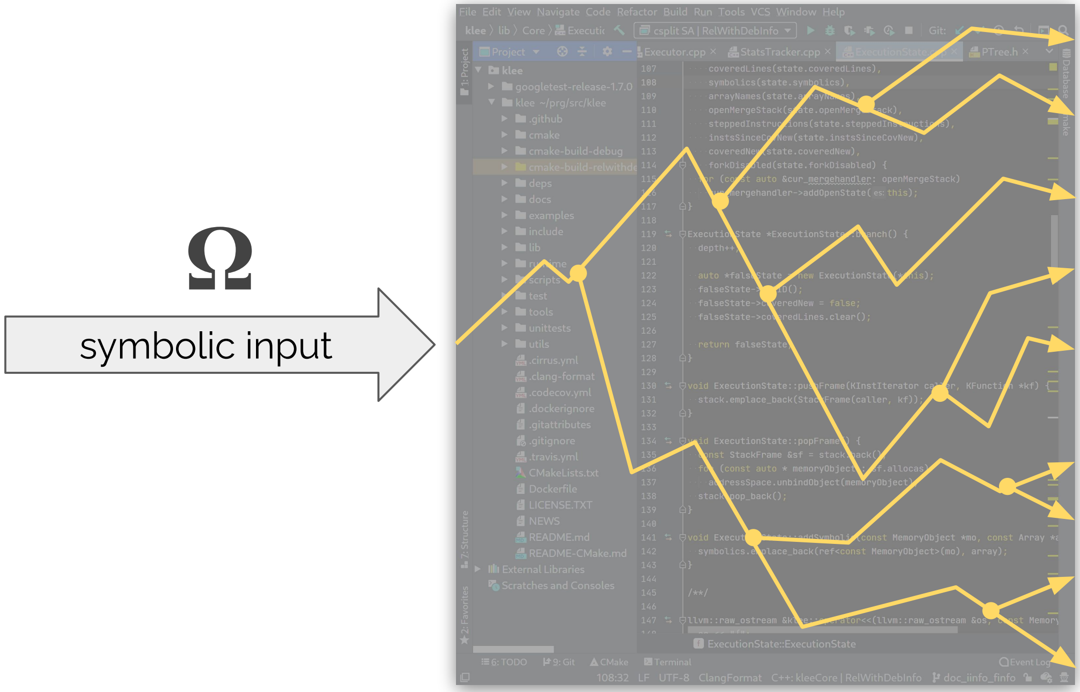
Concrete vs. Symbolic Execution

concrete input



concrete output

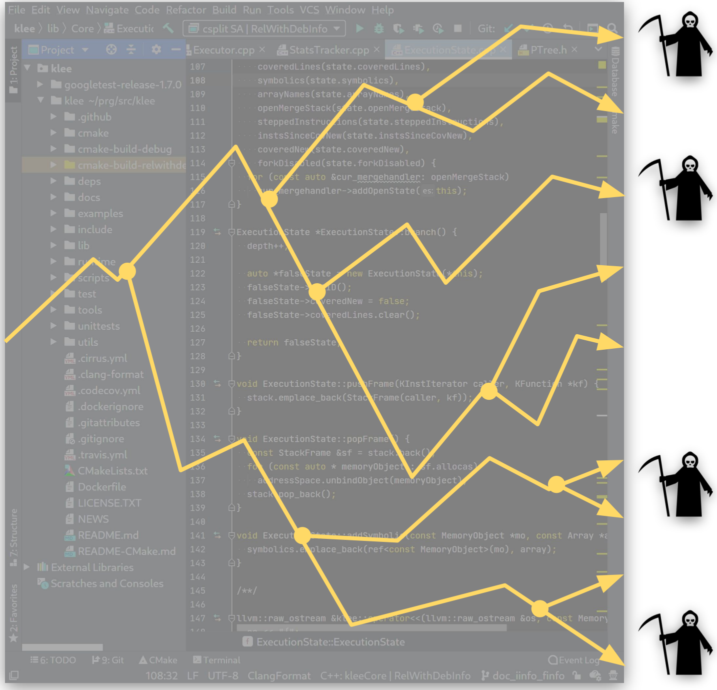
Concrete vs. Symbolic Execution



➔ high-coverage test cases

➔ crashing inputs

Challenges in Symbolic Execution

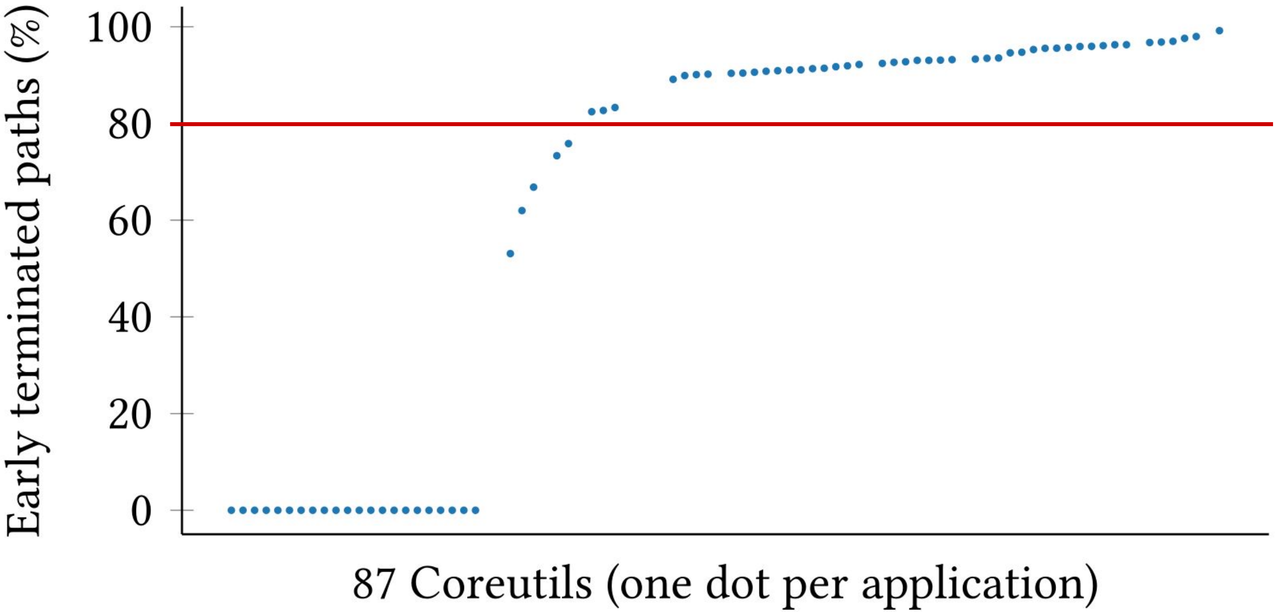


Constraint solving overhead

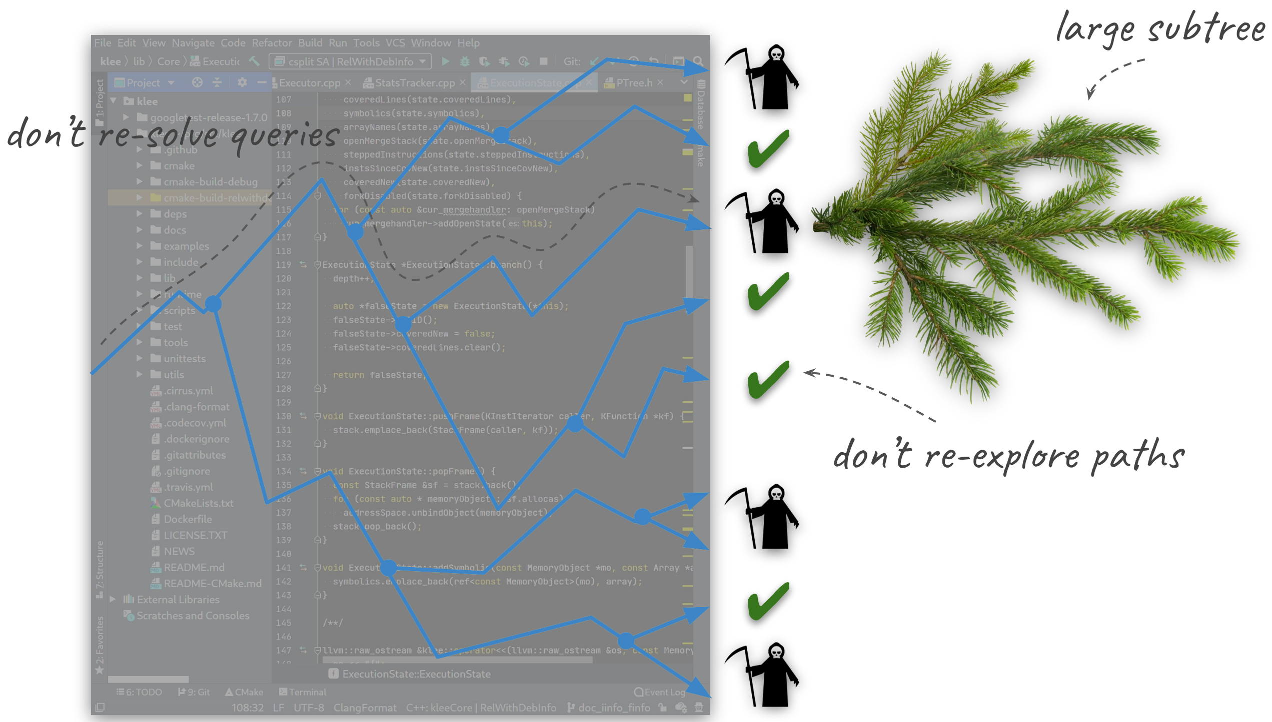
- feasibility checks
- safety checks
- test generation

Path explosion

Early Termination (Memory Pressure)

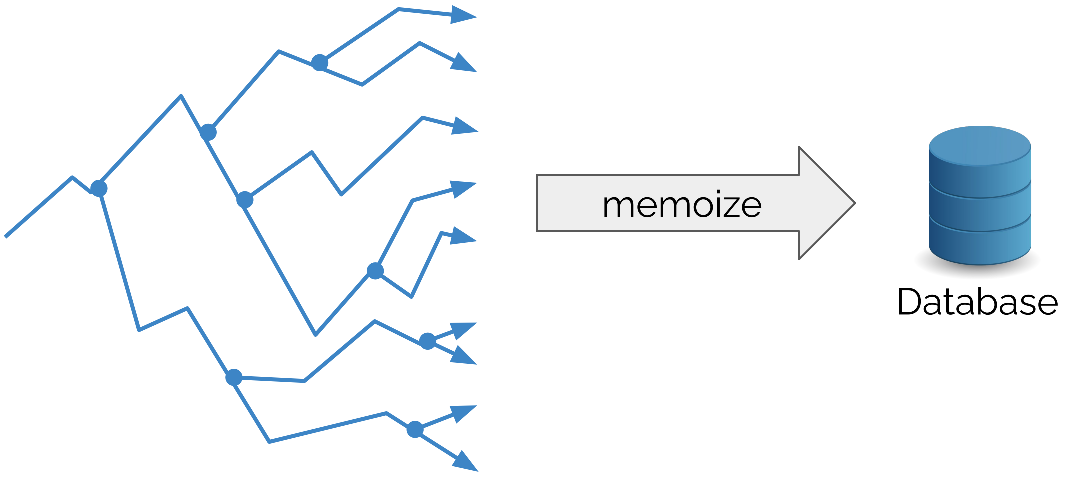


Motivation

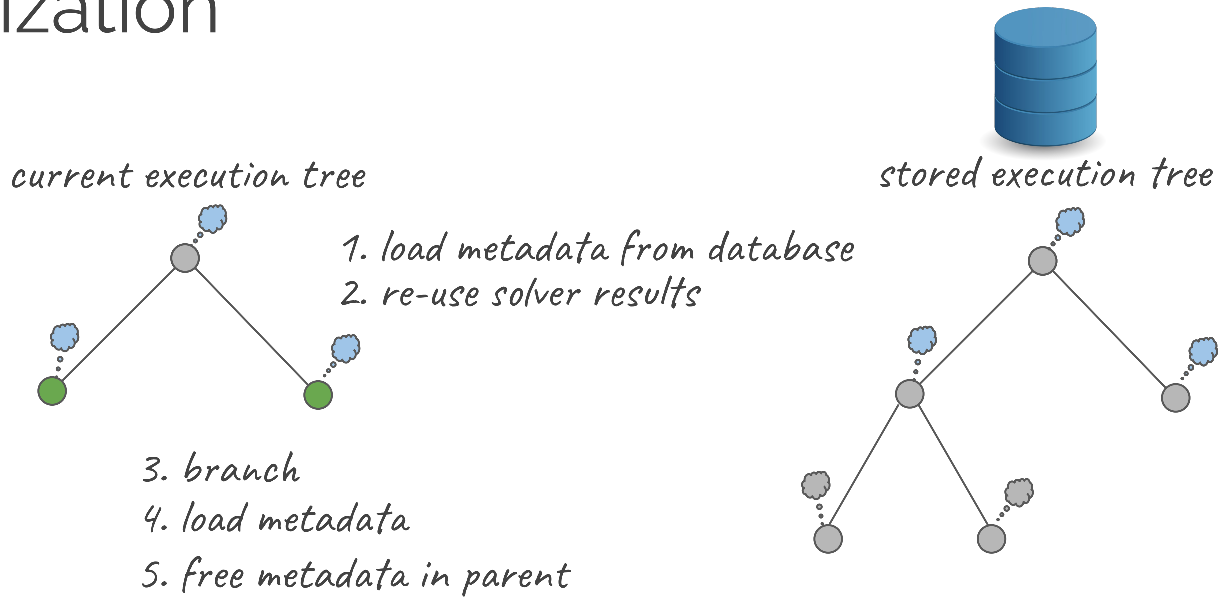


Memoization

- trade time for space
- **store solver results** as metadata in execution tree nodes
- **persist tree to disk**
- **re-use results on re-execution**

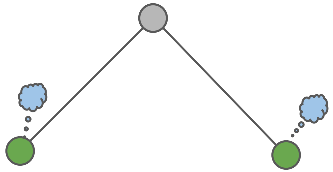


Memoization



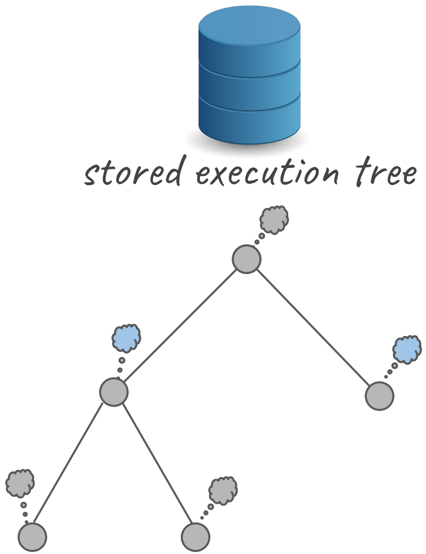
Memoization

current execution tree

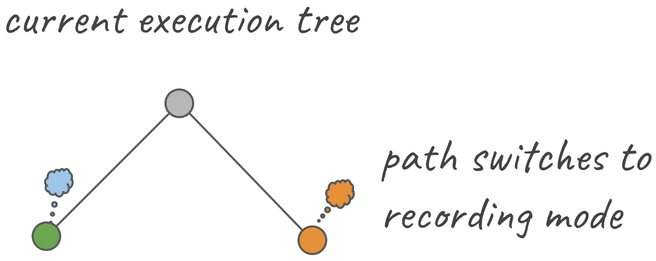


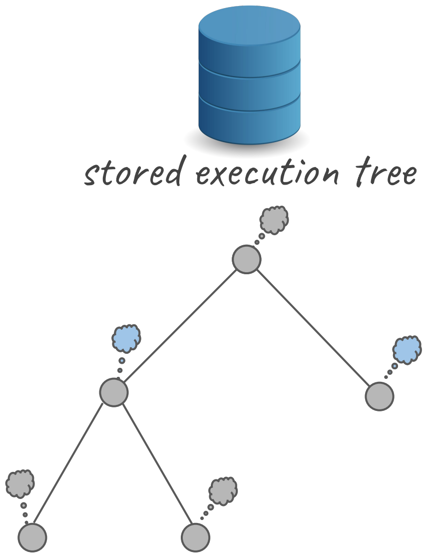
path progresses beyond memoized data

stored execution tree

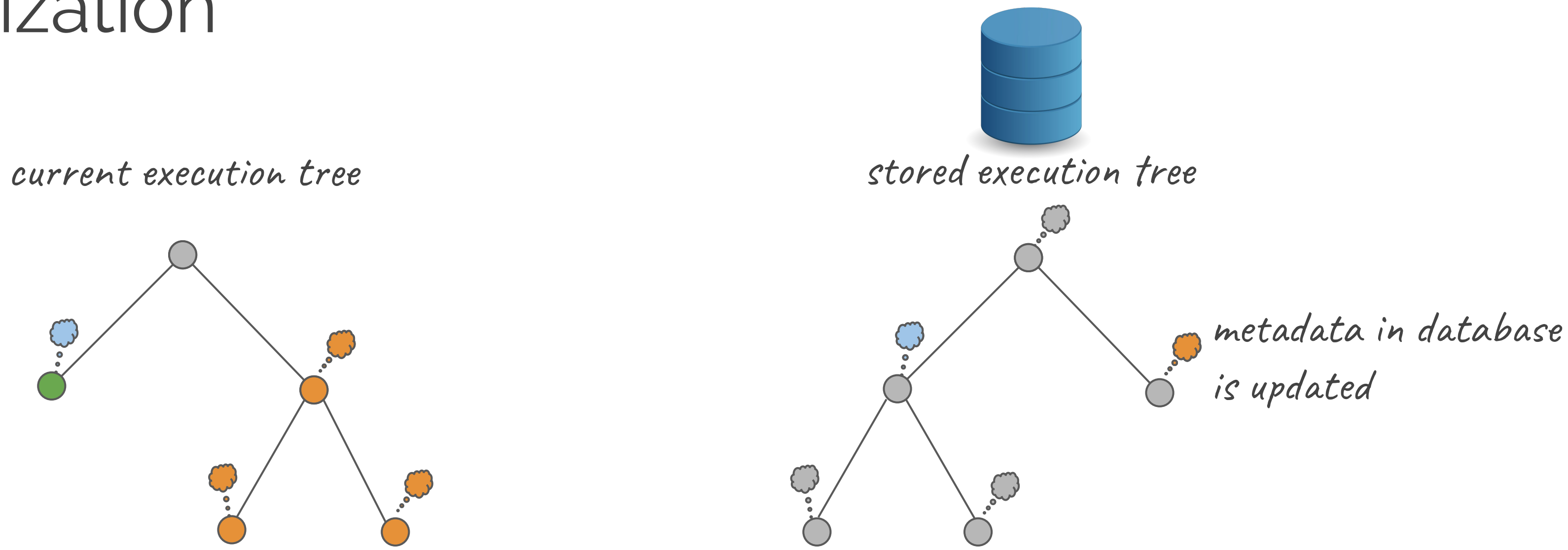


Memoization

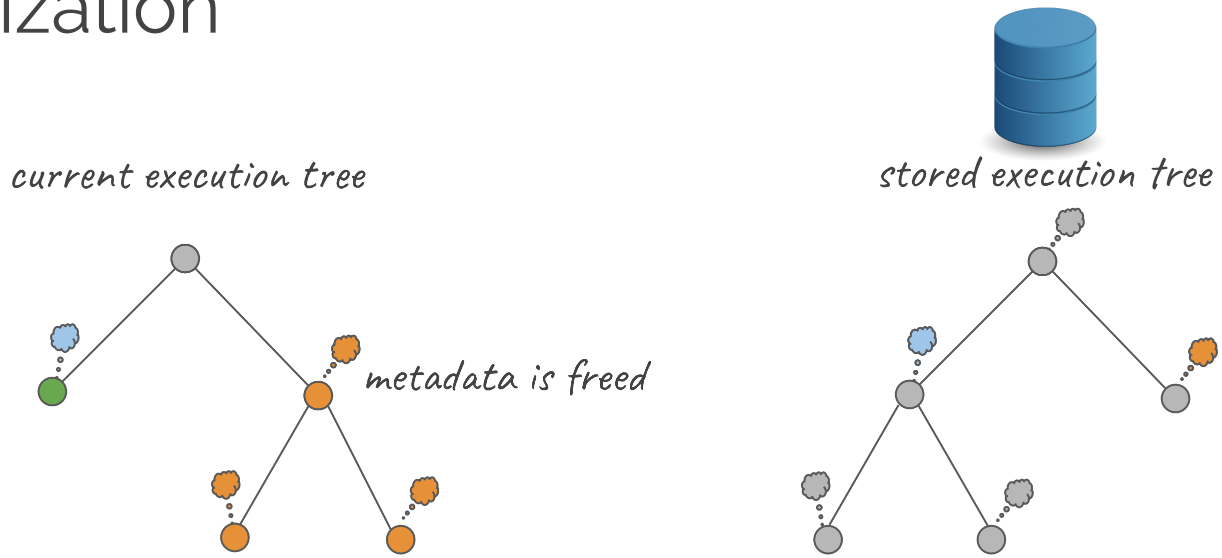




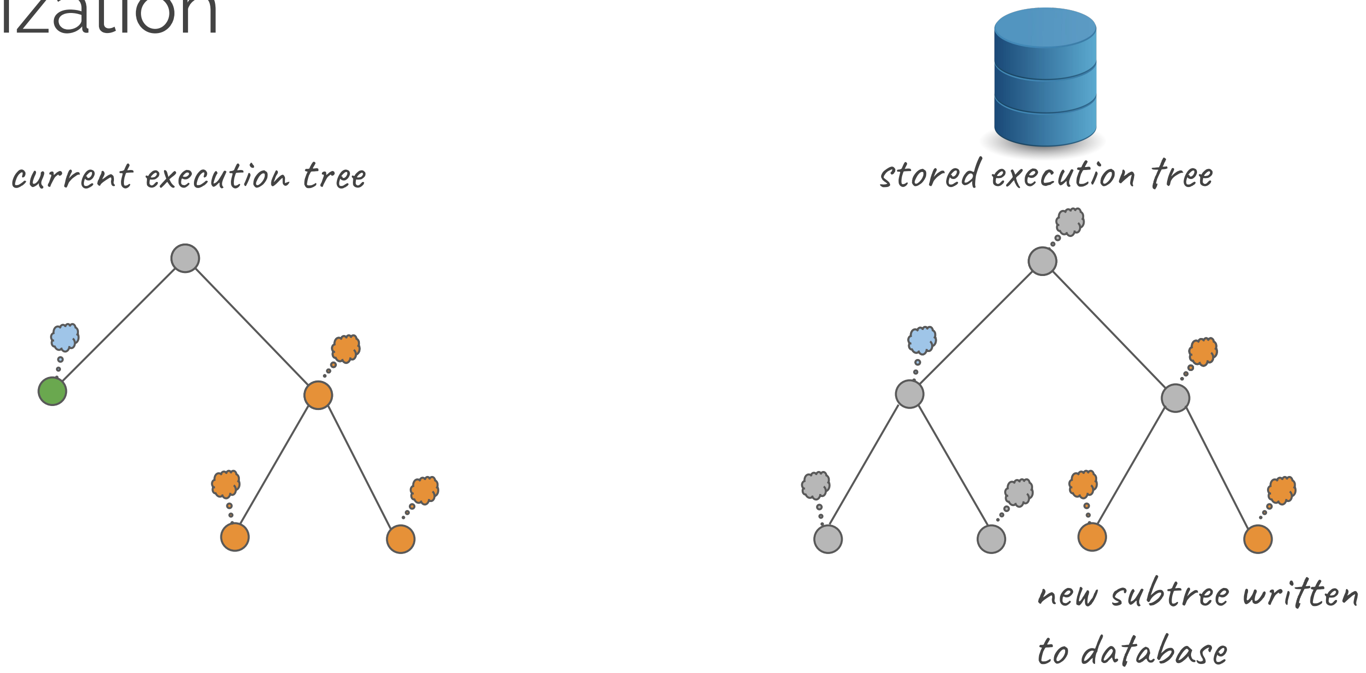
Memoization



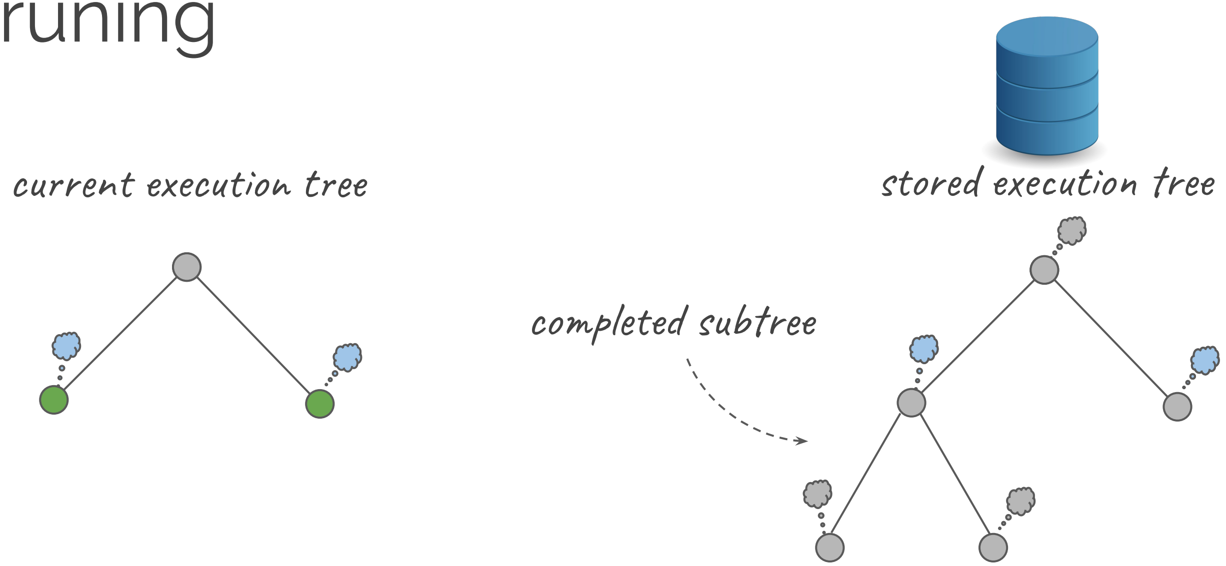
Memoization



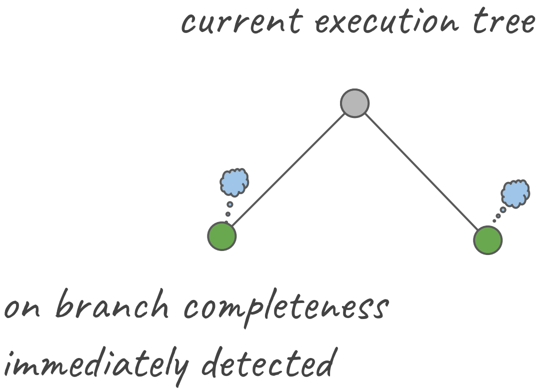
Memoization

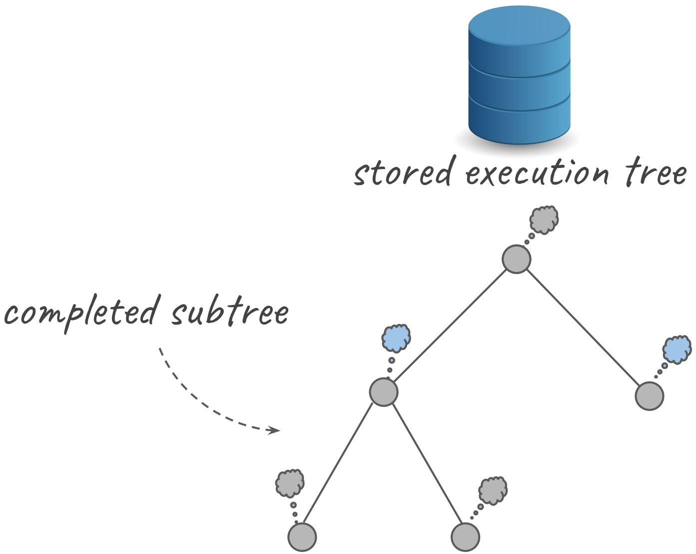


Path Pruning

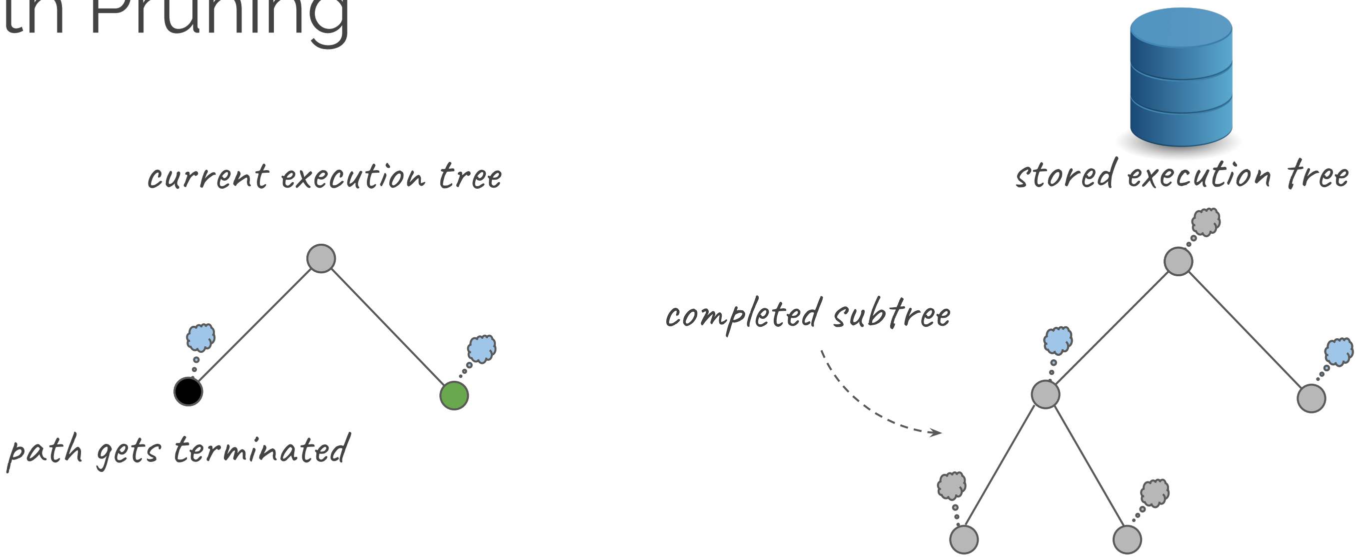


Path Pruning

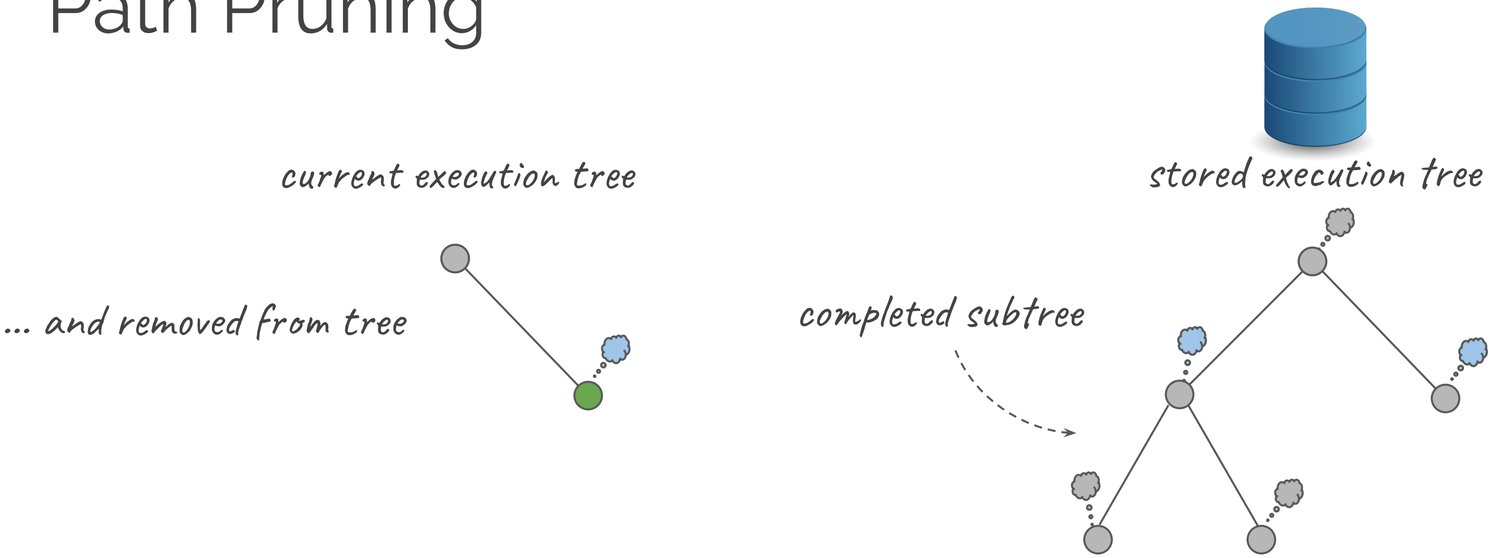




Path Pruning



Path Pruning



ISSTA 2012

Memoized Symbolic Execution

Guowei Yang
University of Texas at Austin
Austin, TX 78712, USA
guoweiyang@utexas.edu

Corina S. Păsăreanu
Carnegie Mellon Silicon Valley
NASA Ames, M/S 269-2
Moffett Field, CA 94035, USA
corina.s.pasareanu@nasa.gov

Sarfraz Khurshid
University of Texas at Austin
Austin, TX 78712, USA
khurshid@ece.utexas.edu

ABSTRACT

This paper introduces *memoized symbolic execution* (*Memoise*), a new approach for more efficient application of forward symbolic execution, which is a well-studied technique for systematic exploration of program behaviors based on bounded execution paths. Our key insight is that application of symbolic execution often requires several successive runs of the technique on largely similar underlying problems, e.g., running it once to check a program to find a bug, fixing the bug, and running it again to check the modified program. Memoise introduces a *trie*-based data structure that stores the key elements of a run of symbolic execution. Maintenance of the trie during successive runs allows re-use of previously computed results of symbolic execution without the need for re-computing them as is traditionally done. Experiments using our prototype implementation of Memoise show the benefits it holds in various standard scenarios of using symbolic execution, e.g., with iterative deepening of exploration depth, to perform regression analysis, or to enhance coverage using heuristics.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Symbolic execution*

Off-the-shelf constraint solvers are used to reason about the formulas to discard those paths whose conditions are unsatisfiable. In practice, the technique can be costly to apply due to its inherent high time and space complexity. There are two key factors that determine its cost: (1) the number of paths that need to be explored and (2) the cost of constraint solving.

Recent years have seen substantial advances in raw computation power and constraint solving technology [1], as well as in basic algorithmic approaches for symbolic execution [4, 25]. These advances have made symbolic execution applicable to a diverse class of programs and enable a range of analyses, including bug finding using automated test generation – a traditional application of this technique – as well as other novel applications, such as program equivalence checking [23], regression analysis [17], and continuous testing [27]. All these applications utilize the same path-based analysis that lies at the heart of symbolic execution. As such, their effectiveness is determined by the two factors that determine the cost of the symbolic execution, and at present, reducing the cost of symbolic execution remains a fundamental challenge.

This paper introduces *memoized symbolic execution* (*Memoise*), a new approach that addresses both factors to enable more efficient applications of symbolic execution. Our key insight is that applying symbolic execution often requires several successive runs of

The second assumption maintains the correspondence of the executions of program paths across different runs of symbolic execution, and makes feasible the reuse of symbolic execution results. As long as the same search order is used during re-execution, the symbolic execution tree corresponding to the same program executions remain the same, and this assures the correctness of trie-guided symbolic execution. Merging is correct since the executions corresponding to the removed parts remain the same in re-execution and will yield to the same sub-trie, and thus the removed p
be brought back from the old trie.

(same search strategy)

3.2.1 *Node Marking*

The first step in memoized execution is to *mark* nodes of *interest*. Specifically, we characterize parts of the old trie that may be updated using *candidate* nodes, which represent roots of sub-trees potentially updated during memoized execution. Given the candidate nodes, we mark nodes on paths that need re-execution – all nodes on any path from the trie root to a *candidate* node are marked (while the rest of the nodes remain unmarked). The exact classification of candidate nodes depends on the particular analysis that is performed. For example, for iterative deepening, the boundary nodes are the candidate nodes (e.g., $n9$ in Figure 3). regression analysis the nodes that are impacted by the program change are considered as candidate ones (the impacted nodes are found by an impact analysis as described in Section 4.1.2). The node marking is reset at the beginning of memoized analysis.

(loads complete tree)

The second assumption maintains the correspondence of the executions of program paths across different runs of symbolic execution, and makes feasible the reuse of symbolic execution results. As
lon
bol
ren
syn
res
and
be

Table 1: Iterative Deepening Results

(same search strategy)

Depth A	Depth B	Sym Exe at Depth A: Time (ss) Reg	Time (ss) ID	Mem (MB) Reg	Mem (MB) ID	Sym Exe at Depth B: States Reg	States ID	#Solver calls Reg	#Solver calls ID	Time (ss) Reg	Time (ss) ID-p	Time (ss) ID-c	Mem (MB) Reg	Mem (MB) ID-p	Mem (MB) ID-c	Trie (MB) ID-p	Trie (MB) ID-c
24	25	16	21	305	419	349272	252952	335358	77312	20	24	22	242	474	474	18.8	13.4
29	30	34	60	246	260	644184	171784	629758	32256	37	36	27	214	500	486	35.4	9.3

(a) WBS Example

Depth A	Depth B	Sym Exe at Depth A: Time (ss) Reg	Time (ss) ID	Mem (MB) Reg	Mem (MB) ID	Sym Exe at Depth B: States Reg	States ID	#Solver calls Reg	#Solver calls ID	Time (ss) Reg	Time (ss) ID-p	Time (ss) ID-c	Mem (MB) Reg	Mem (MB) ID-p	Mem (MB) ID-c	Trie (MB) ID-p	Trie (MB) ID-c
24	25	35	38	304	367	17103	16756	12252	2942	47	46	45	413	263	395	0.9	0.8
29	30	86	87	419	333	33273	15250	25684	1540	92	45	45	413	345	263	2.0	0.9
34	35	96	97	419	345	35359	1476	27636	18	102	9	9	292	404	243	2.1	0.1

(b) MerArbiter Example

(loads complete tree)

Depth A	Depth B	Sym Exe at Depth A: Time (ss) Reg	Time (ss) ID	Mem (MB) Reg	Mem (MB) ID	Sym Exe at Depth B: States Reg	States ID	#Solver calls Reg	#Solver calls ID	Time (ss) Reg	Time (ss) ID-p	Time (ss) ID-c	Mem (MB) Reg	Mem (MB) ID-p	Mem (MB) ID-c	Trie (MB) ID-p	Trie (MB) ID-c
9	10	127	125	425	352	674	647	255	121	195	76	77	421	343	296	0.03	0.03
11	12	538	549	413	414	2243	2113	2160	966	1033	490	483	390	316	420	0.12	0.11

(c) Apollo Example

er-
be
ees
an-
all
ted
ifi-
hat
ary
on
are
considered as candidate ones (the impacted nodes are found by an impact analysis as described in Section 4.1.2). The node marking is reset at the beginning of memoized analysis.

(short runtimes)
No divergence detection.

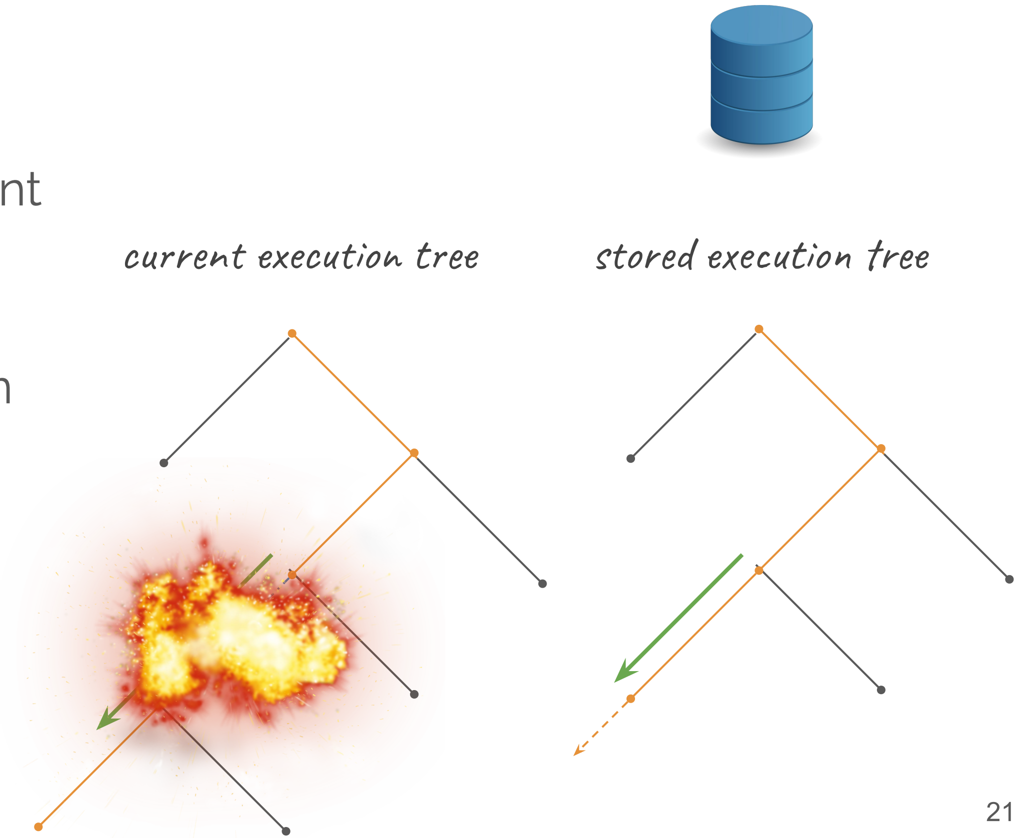
Divergences

Causes

- changes in external environment (disk layout, date, environment variables)
- shared address space between execution states

Problem

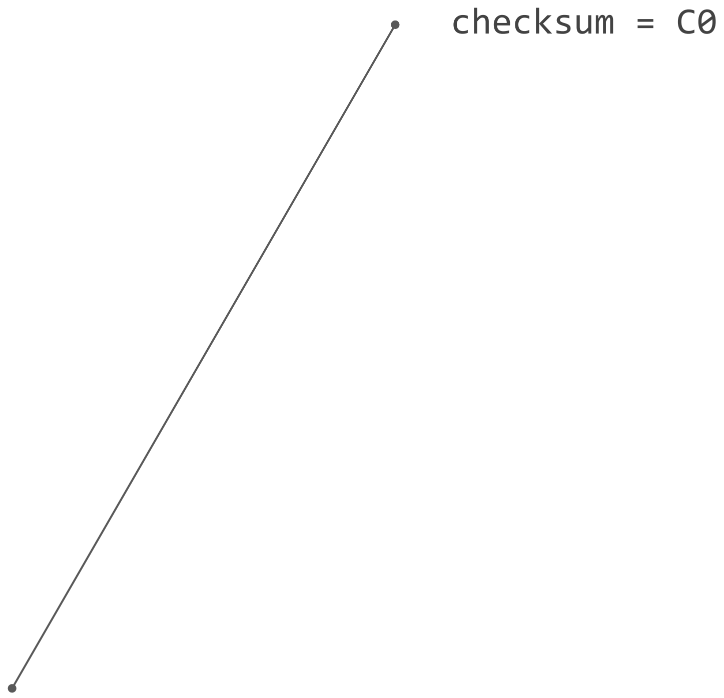
- exploration of infeasible paths
- false negatives



Divergence Detection

Mitigation

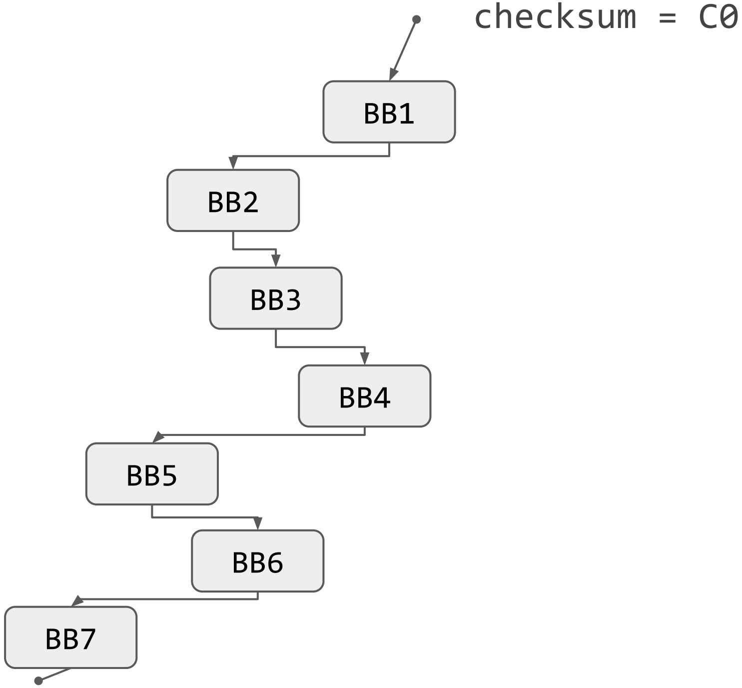
- checksum over sequence of basic blocks validated on each branch
- affected paths are reset



Divergence Detection

Mitigation

- checksum over sequence of basic blocks validated on each branch
- affected paths are reset

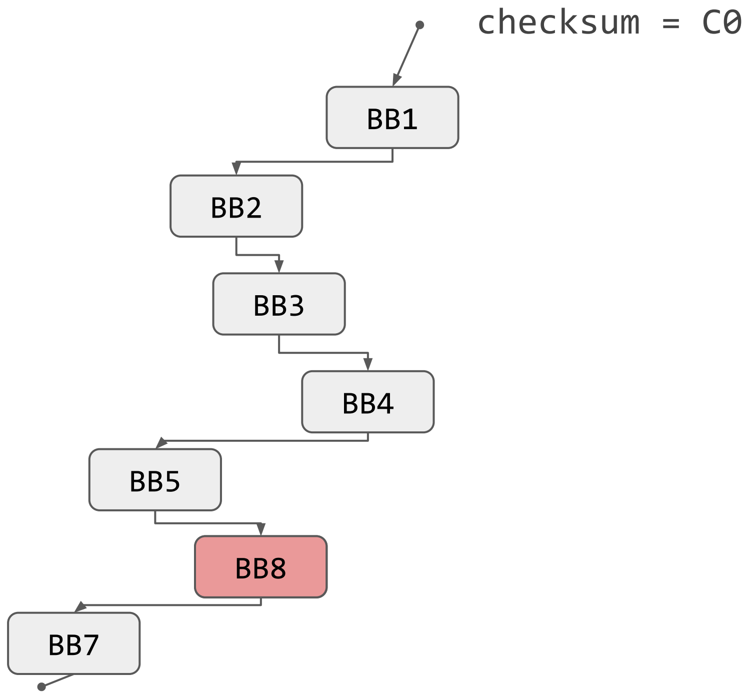


`checksum = hash(BB7) ⊗ hash(BB6) ⊗ ... ⊗ C0`

Divergence Detection

Mitigation

- checksum over sequence of basic blocks validated on each branch
- affected paths are reset

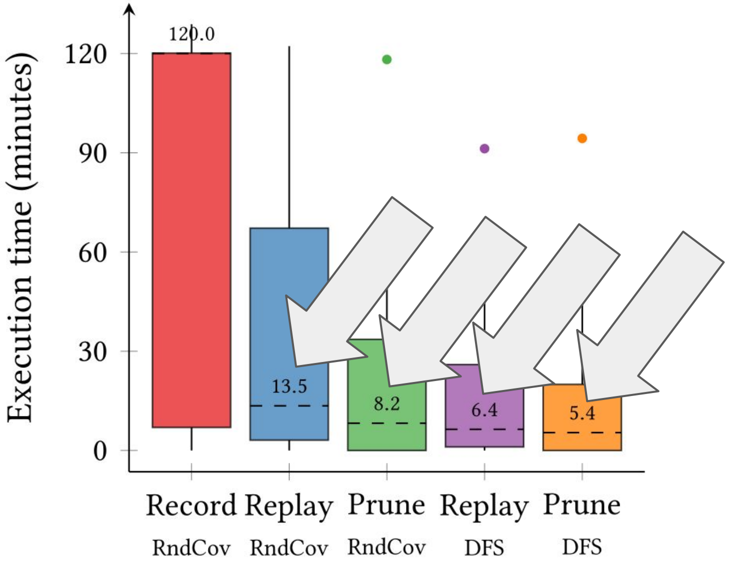


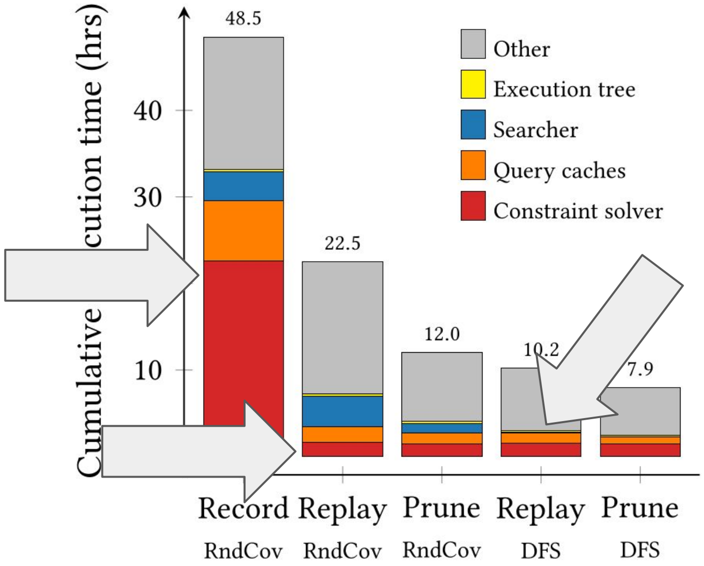
`checksum ≠ hash(BB7) ⊗ hash(BB8) ⊗ ... ⊗ C0`

Evaluation

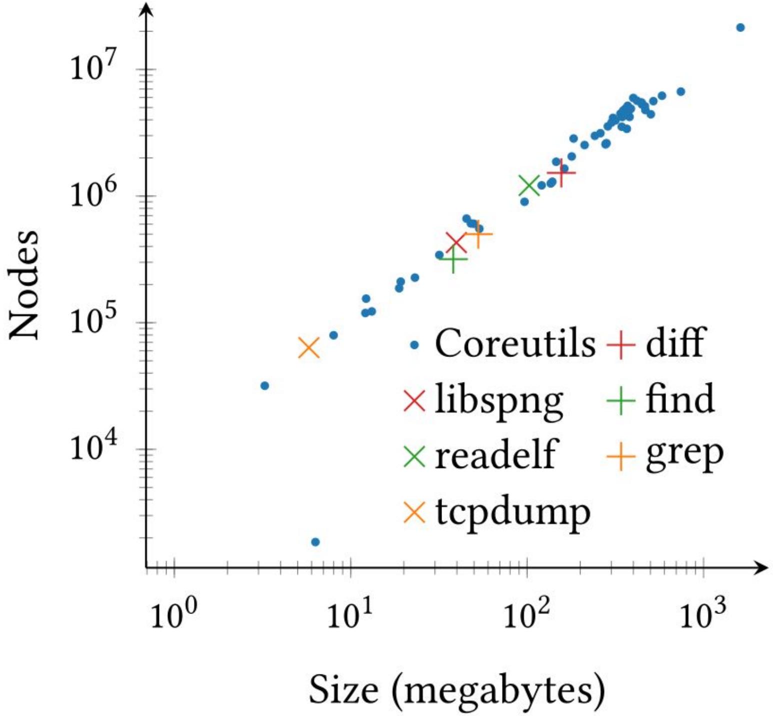
- MoKlee is implemented on top of KLEE 1.4
- evaluated on 93 benchmarks:
  - readelf (Binutils)
  - 87 Coreutils
  - diff (Diffutils)
  - find (Findutils)
  - grep
  - libspng
  - tcpdump

Evaluation - Runtime

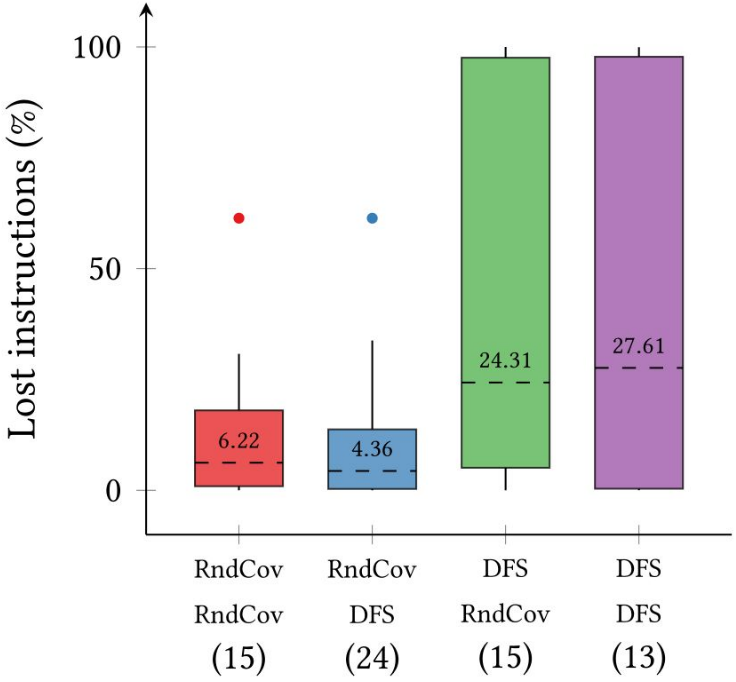




Evaluation - Storage Size



Evaluation - Divergences



Evaluation - Long Running Symbolic Execution

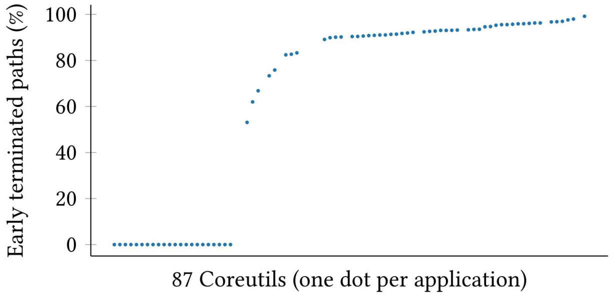
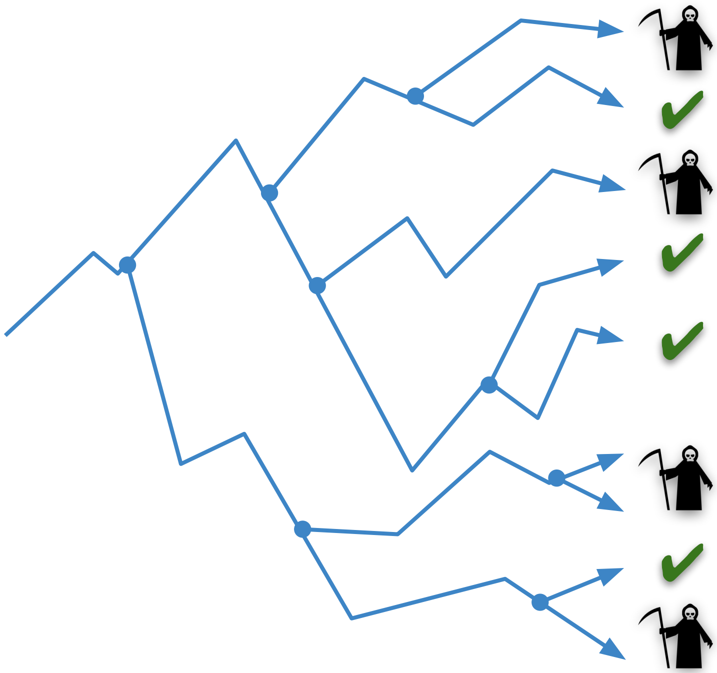
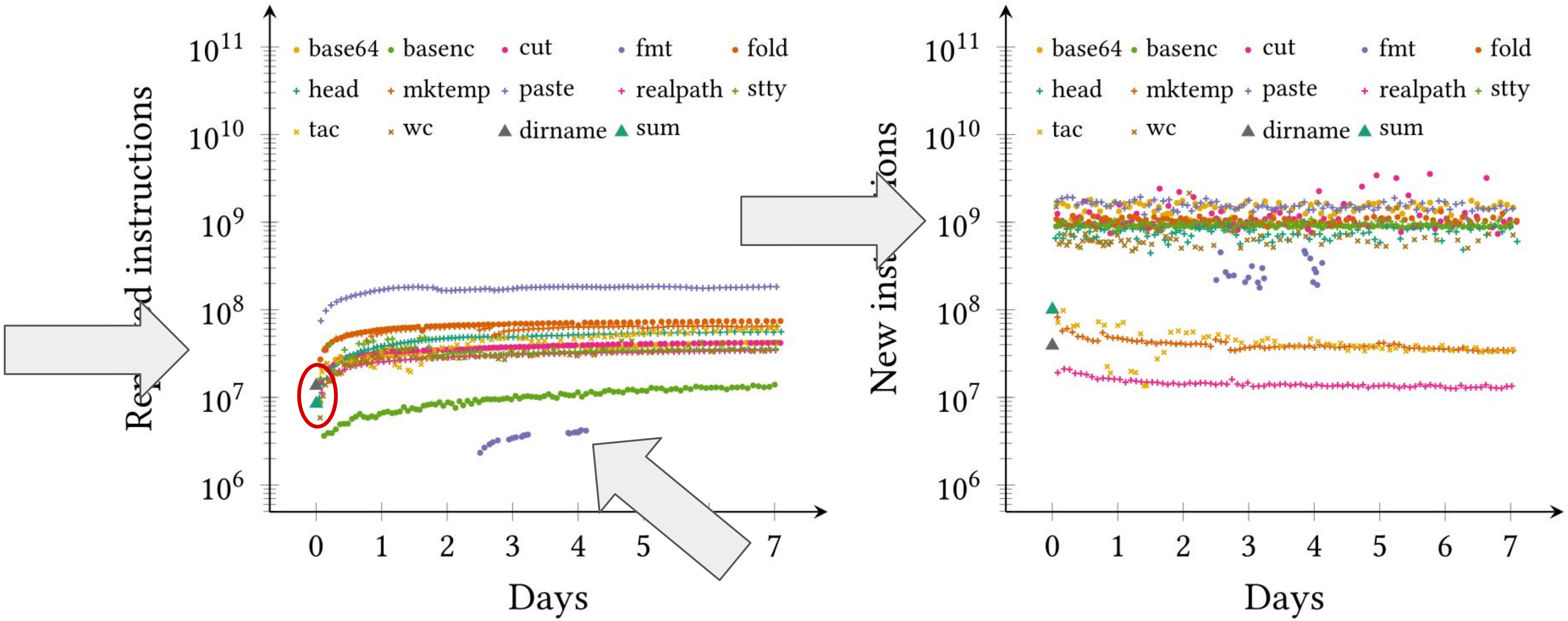


Figure 1: When running KLEE[1] on 87 *Coreutils* for 2 h each with the default search heuristic and memory limit (2 GB), most paths are terminated early due to memory pressure.

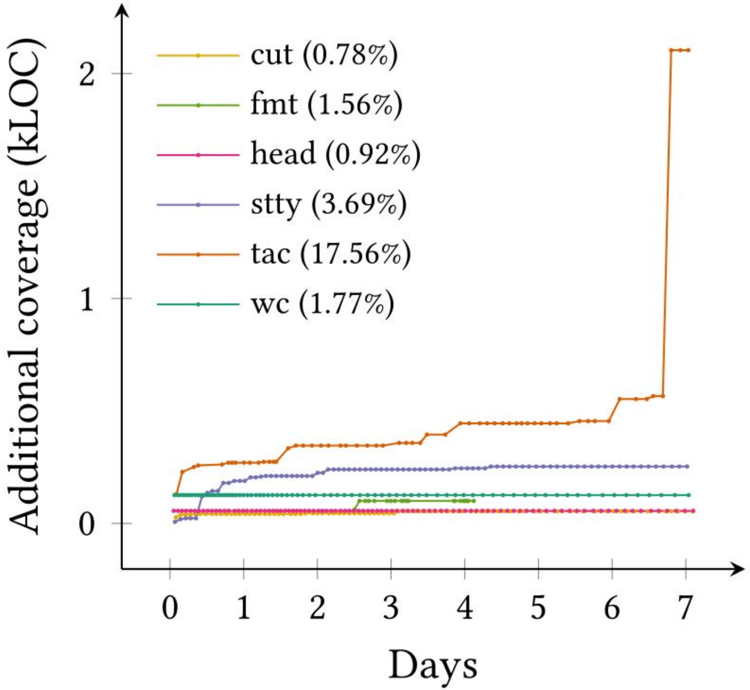


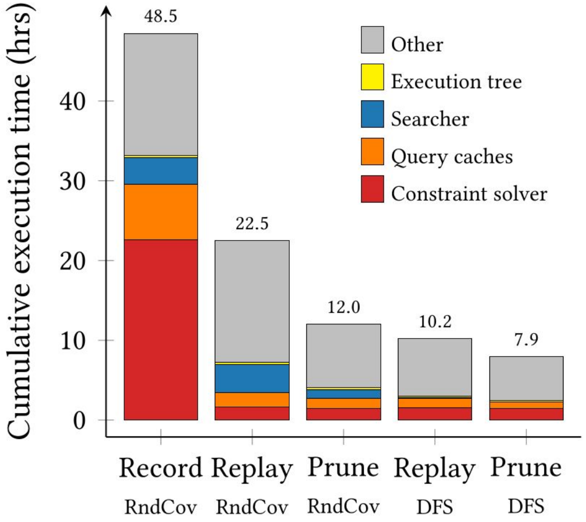
14 applications terminate ran out of states before the 2h limit!

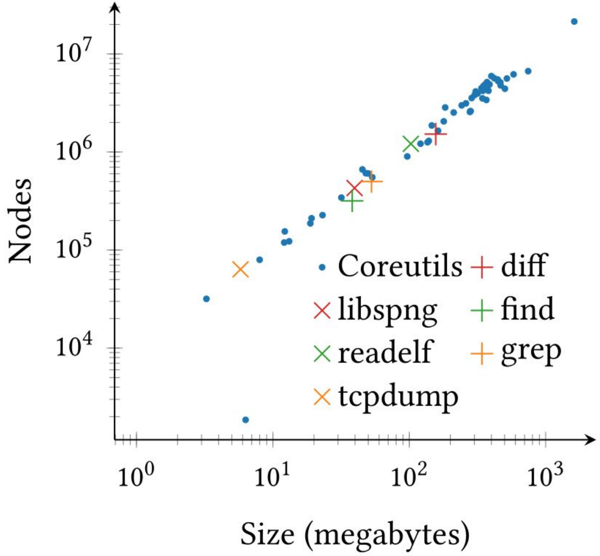
Evaluation - Long Running Symbolic Execution

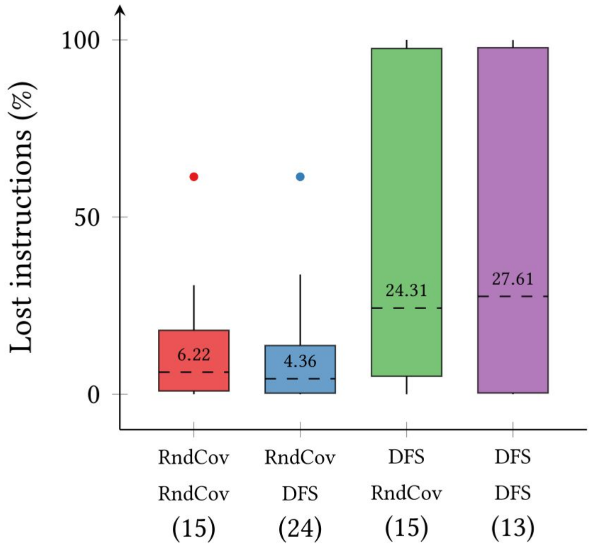


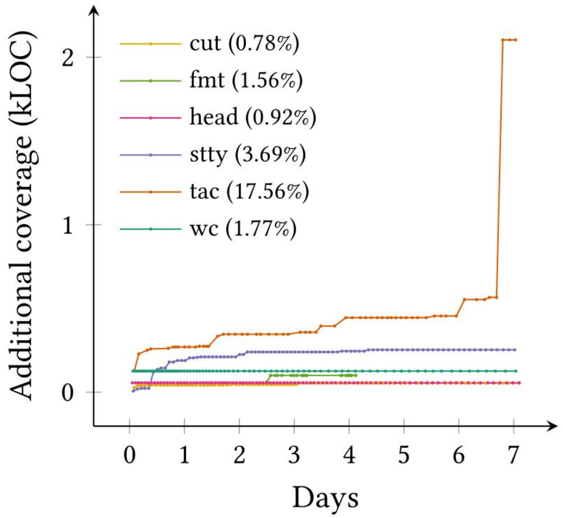
Evaluation - Long Running Symbolic Execution











MoKlee Artefact: https://srg.doc.ic.ac.uk/projects/moklee/

KLEE: https://klee.github.io/

2nd KLEE Workshop: 22-23 April 2021 in London