SOFTWARE RELIABILITY
GROUP

Running Symbolic Execution Forever

Frank Busse · Martin Nowack · Cristian Cadar
Imperial College London

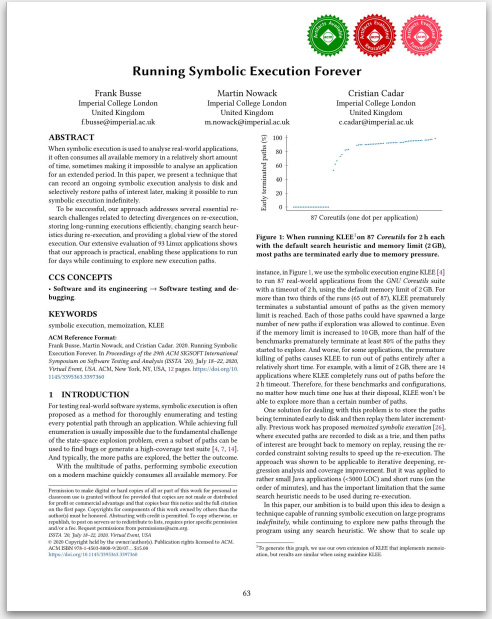
2nd International KLEE Workshop on Symbolic Execution
10-11 June 2021

MoKlee (ISSTA 2020)

Project: https://srg.doc.ic.ac.uk/projects/moklee/

Talk: https://youtu.be/KNEwLszhuuA

Artefact: https://zenodo.org/record/3895271



Running Symbolic Execution Forever

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

When symbolic execution is used to analyse real-world applications, it often consumes all available memory in a relatively short amount of time, sometimes making it impossible to analyse an application for an extended period. In this paper, we present a technique that can record an ongoing symbolic execution analysis to disk and selectively restore paths of interest later, making it possible to run symbolic execution indefinitely.

To be successful, our approach addresses several essential research challenges related to detecting divergences on re-execution, storing long-running executions efficiently, changing search heuristics during re-execution, and providing a global view of the stored execution. Our extensive evaluation of 93 Linux applications shows that our approach is practical, enabling these applications to run for days while continuing to explore new execution paths.

CCS CONCEPTS

• Software and its engineering → Software testing and debugging.

KEYWORDS

symbolic execution, memoization, KLEE

ACM Reference Format:

Frank Busse, Martin Nowack, and Cristian Cadar. 2020. Running Symbolic Execution Forever. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3395363.3397360

1 INTRODUCTION

For testing real-world software systems, symbolic execution is often proposed as a method for thoroughly enumerating and testing every potential path through an application. While achieving full enumeration is usually impossible due to the fundamental challenge of the state-space explosion problem, even a subset of paths can be used to find bugs or generate a high-coverage test suite [4, 7, 14]. And typically, the more paths are explored, the better the outcome.

With the multitude of paths, performing symbolic execution on a modern machine quickly consumes all available memory. For instance, in Figure 1, we use the symbolic execution engine KLEE [4] to run 87 real-world applications from the *GNU Coreutils* suite with a timeout of 2 h, using the default memory limit of 2 GB. For more than two thirds of the runs (65 out of 87), KLEE prematurely terminates a substantial amount of paths as the given memory limit is reached. Each of those paths could have spawned a large number of new paths if exploration was allowed to continue. Even if the memory limit is increased to 10 GB, more than half of the benchmarks prematurely terminate at least 80% of the paths they started to explore. And worse, for some applications, the premature killing of paths causes KLEE to run out of paths entirely after a relatively short time. For example, with a limit of 2 GB, there are 14 applications where KLEE completely runs out of paths before the 2 h timeout. Therefore, for these benchmarks and configurations, no matter how much time one has at their disposal, KLEE won't be able to explore more than a certain number of paths.

One solution for dealing with this problem is to store the paths being terminated early to disk and then replay them later incrementally. Previous work has proposed *memoized symbolic execution* [26], where executed paths are recorded to disk as a trie, and then paths of interest are brought back to memory on replay, reusing the recorded constraint solving results to speed up the re-execution. The approach was shown to be applicable to iterative deepening, regression analysis and coverage improvement. But it was applied to rather small Java applications (<5000 LOC) and short runs (on the order of minutes), and has the important limitation that the same search heuristic needs to be used during re-execution.

In this paper, our ambition is to build upon this idea to design a technique capable of running symbolic execution on large programs *indefinitely*, while continuing to explore new paths through the program using any search heuristic. We show that to scale up

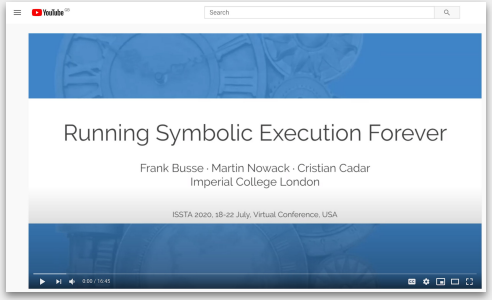
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ISSTA '20, July 18–22, 2020, Virtual Event, USA
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8008-9/20/07…$15.00
https://doi.org/10.1145/3395363.3397360

Early terminated paths (%)
100
80
60
40
20
0
87 Coreutils (one dot per application)

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.

[1] To generate this graph, we use our own extension of KLEE that implements memoization, but results are similar when using mainline KLEE.

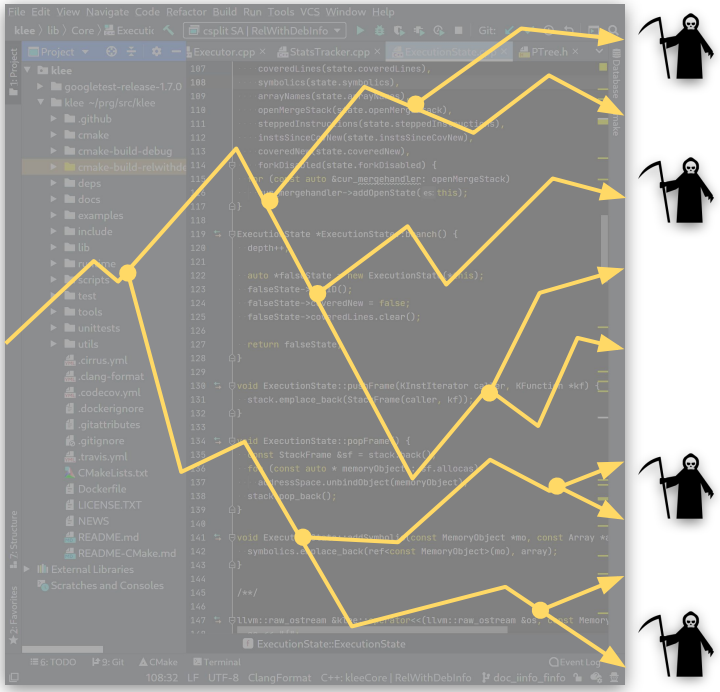
63



Challenges in Symbolic Execution

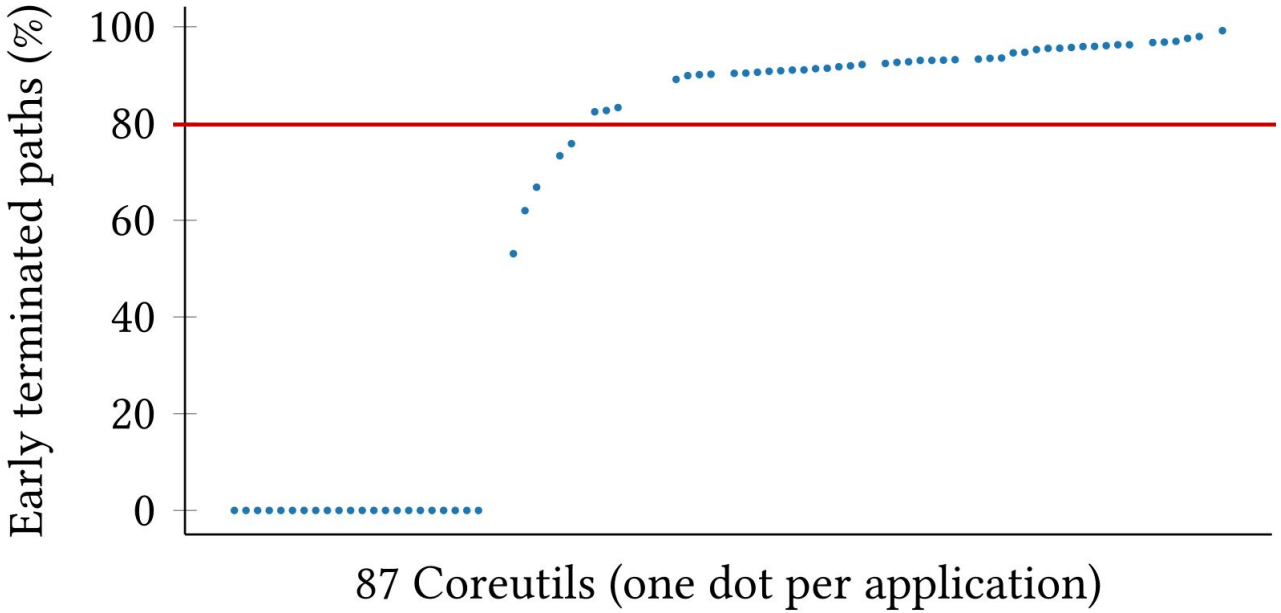
Constraint solving overhead

- feasibility checks
- safety checks
- test generation

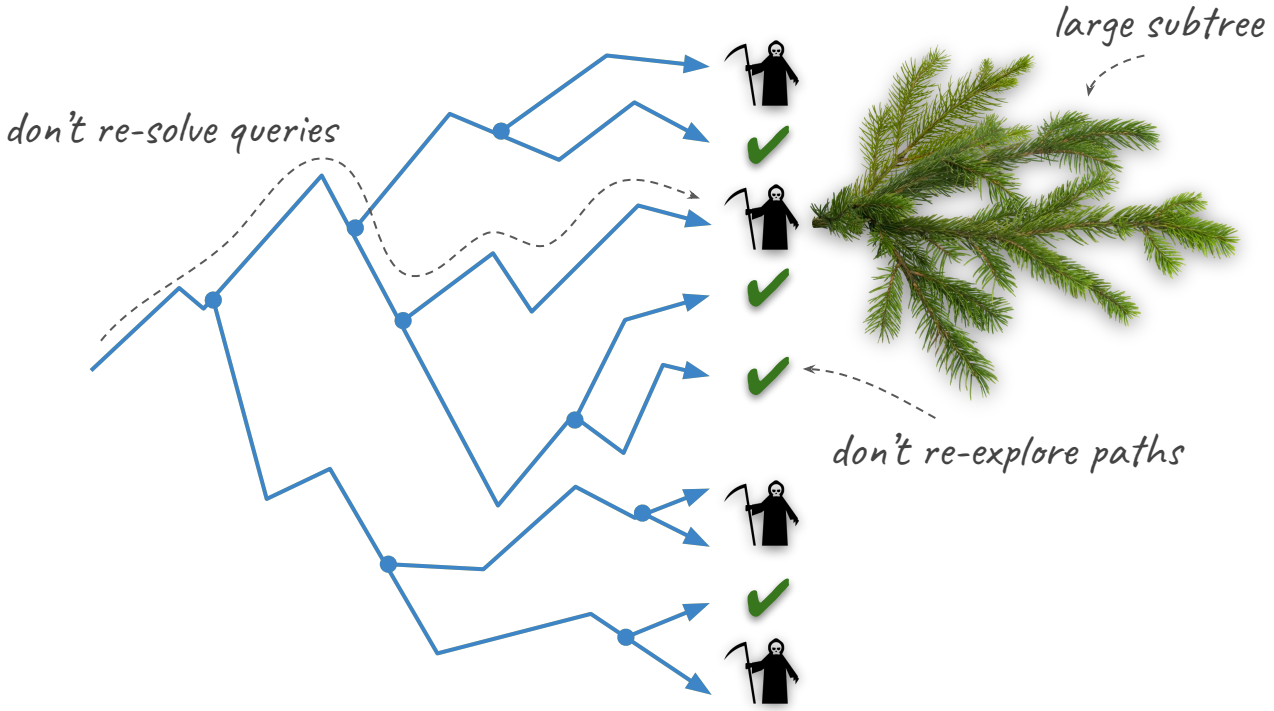


Path explosion

Early Termination (Memory Pressure)

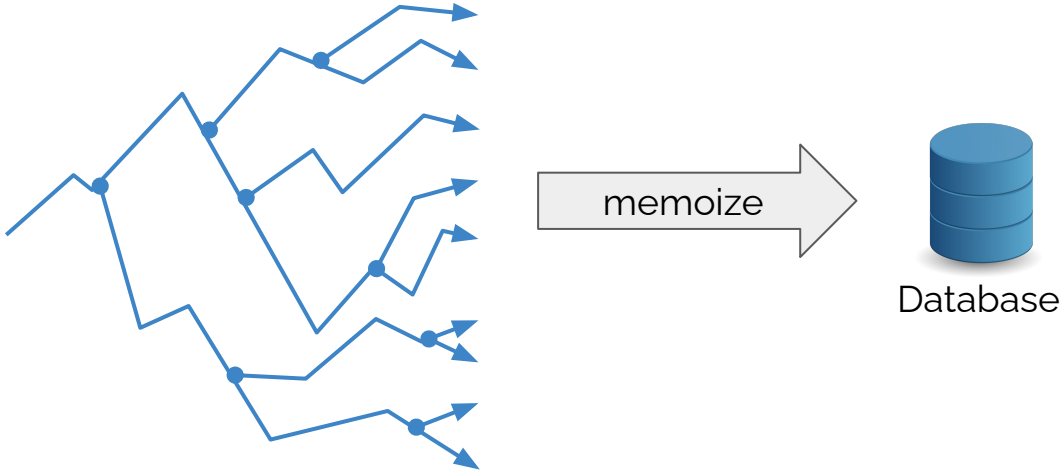


Motivation

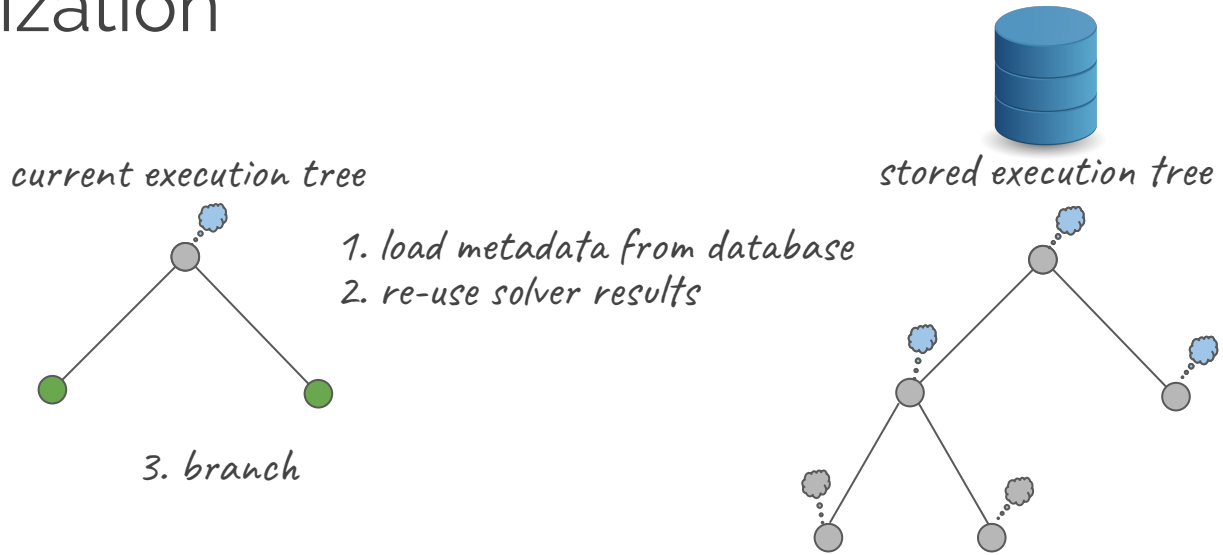


Memoization

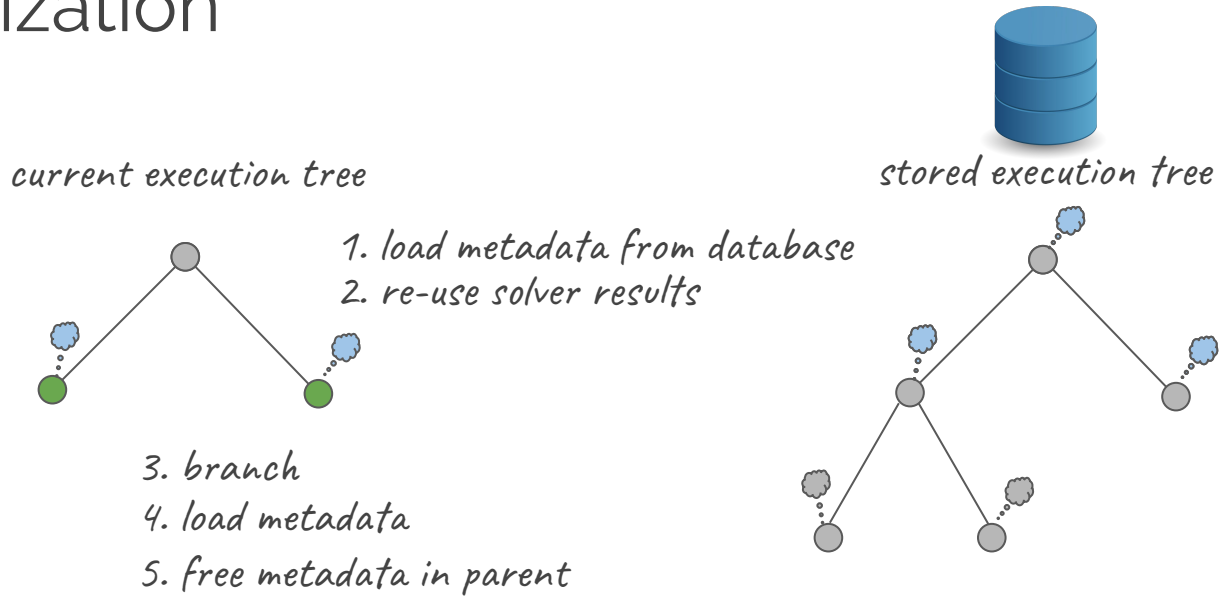
- trade time for space
- **store solver results** as metadata in execution tree nodes
- **persist tree to disk**
- **re-use results on re-execution**



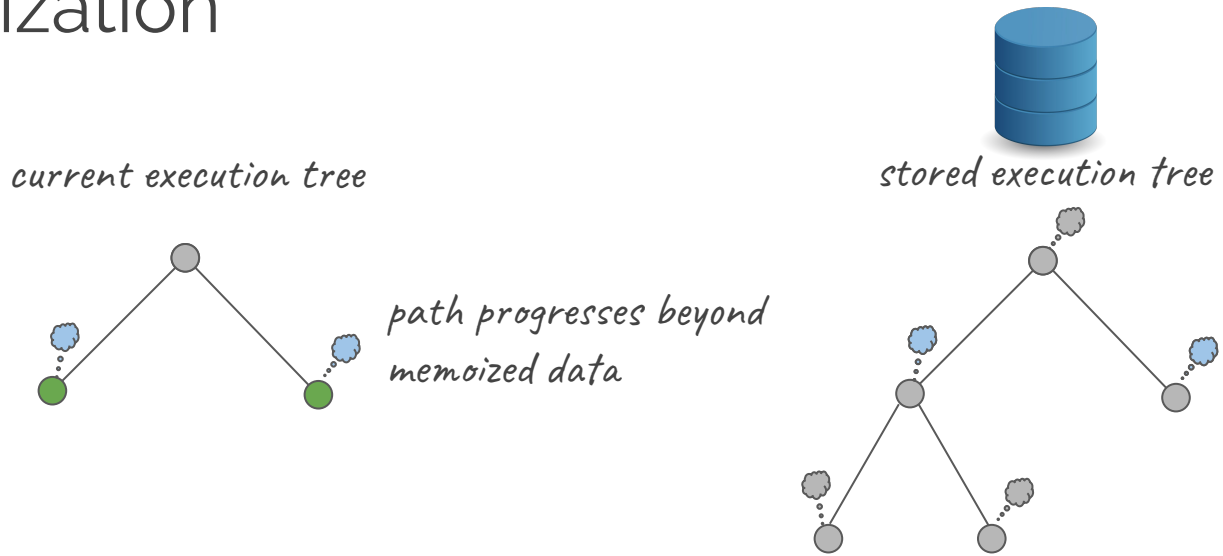
Memoization



Memoization

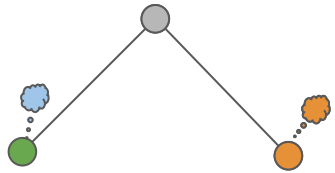


Memoization



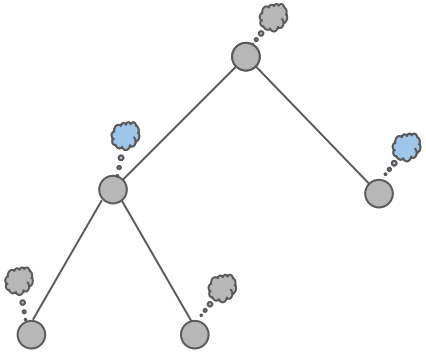
Memoization

current execution tree

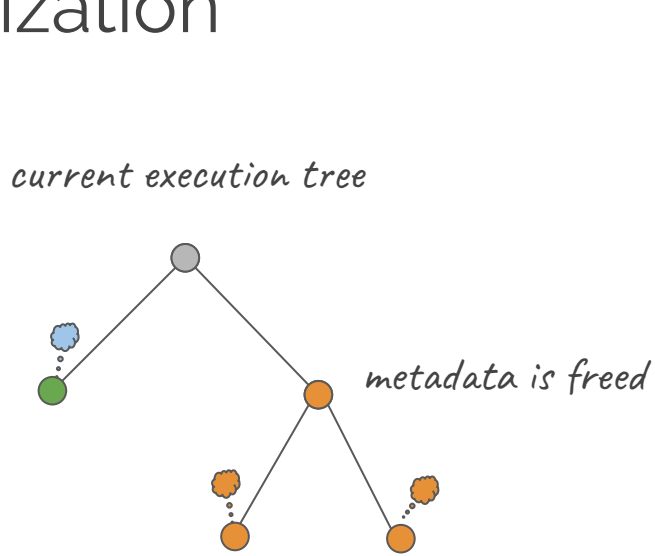


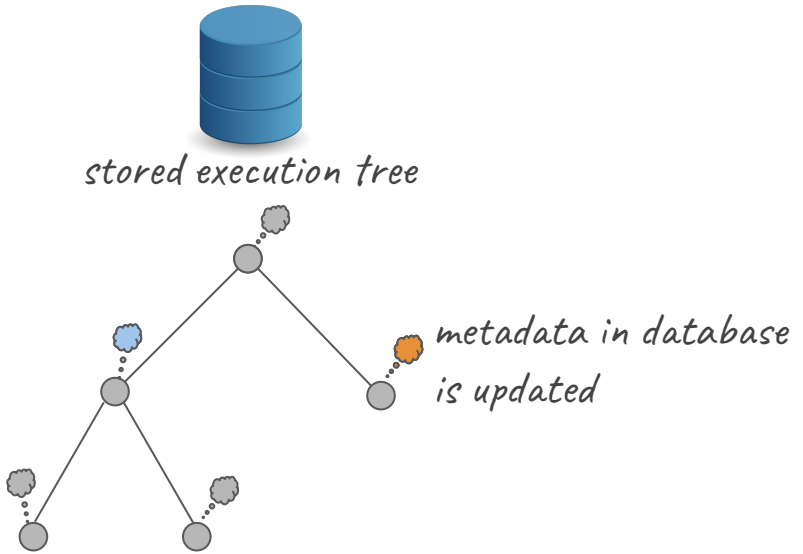
path switches to recording mode

stored execution tree

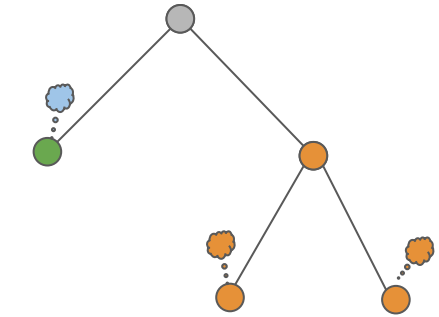


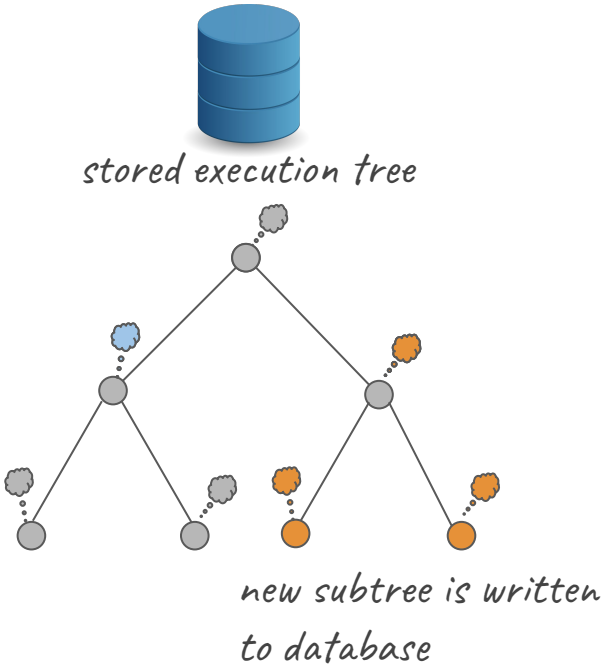
Memoization





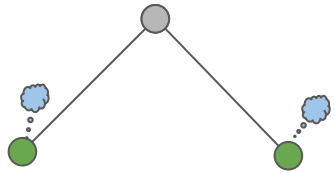
Memoization





Path Pruning

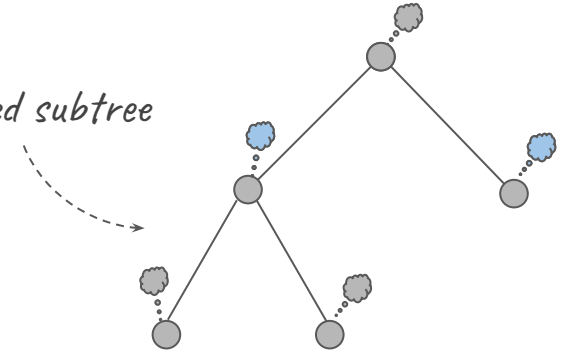
current execution tree



on branch completeness
immediately detected

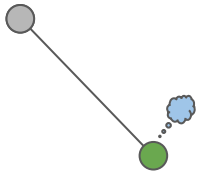
stored execution tree

completed subtree



Path Pruning

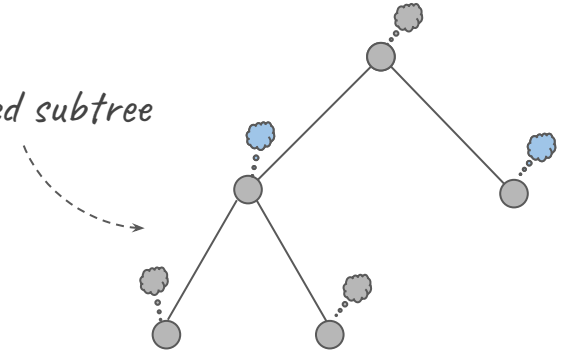
current execution tree



path gets terminated and removed from tree

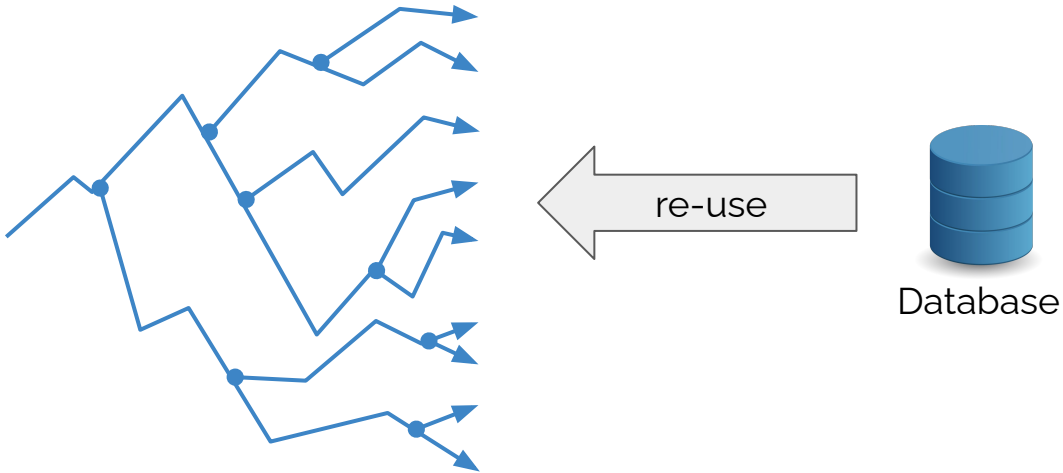
stored execution tree

completed subtree



Persistent Execution Tree (Process Tree)

- shape-analysis (depth, width) to compare search strategies
- compare different executions (deterministic experiments)
- replay/debug single paths w\o test case
- ...



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 recomputing 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 parts can be brought back from the old trie.

(same search strategy)

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.

(loads complete tree)

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.

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

Table 1: Iterative Deepening Results

Depth		Sym Exe at Depth A				Sym Exe at Depth B										
		Time (ss)		Mem (MB)		States		#Solver calls		Time (ss)			Mem (MB)			Trie (MB)	
A	B	Reg	ID	Reg	ID	Reg	ID	Reg	ID	Reg	ID-p	ID-c	Reg	ID-p	ID-c	ID-p	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		Sym Exe at Depth A				Sym Exe at Depth B										
		Time (ss)		Mem (MB)		States		#Solver calls		Time (ss)			Mem (MB)			Trie (MB)	
A	B	Reg	ID	Reg	ID	Reg	ID	Reg	ID	Reg	ID-p	ID-c	Reg	ID-p	ID-c	ID-p	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

Depth		Sym Exe at Depth A				Sym Exe at Depth B										
		Time (ss)		Mem (MB)		States		#Solver calls		Time (ss)			Mem (MB)			Trie (MB)	
A	B	Reg	ID	Reg	ID	Reg	ID	Reg	ID	Reg	ID-p	ID-c	Reg	ID-p	ID-c	ID-p	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

(short runtimes)

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.

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

Table 1: Iterative Deepening Results

Depth		Sym Exe at Depth A				Sym Exe at Depth B										
		Time (ss)		Mem (MB)		States		#Solver calls		Time (ss)			Mem (MB)			Trie (MB)	
A	B	Reg	ID	Reg	ID	Reg	ID	Reg	ID	Reg	ID-p	ID-c	Reg	ID-p	ID-c	ID-p	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		Sym Exe at Depth A				Sym Exe at Depth B										
		Time (ss)		Mem (MB)		States		#Solver calls		Time (ss)			Mem (MB)			Trie (MB)	
A	B	Reg	ID	Reg	ID	Reg	ID	Reg	ID	Reg	ID-p	ID-c	Reg	ID-p	ID-c	ID-p	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

Depth		Sym Exe at Depth A				Sym Exe at Depth B										
		Time (ss)		Mem (MB)		States		#Solver calls		Time (ss)			Mem (MB)			Trie (MB)	
A	B	Reg	ID	Reg	ID	Reg	ID	Reg	ID	Reg	ID-p	ID-c	Reg	ID-p	ID-c	ID-p	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

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.

No divergence detection.

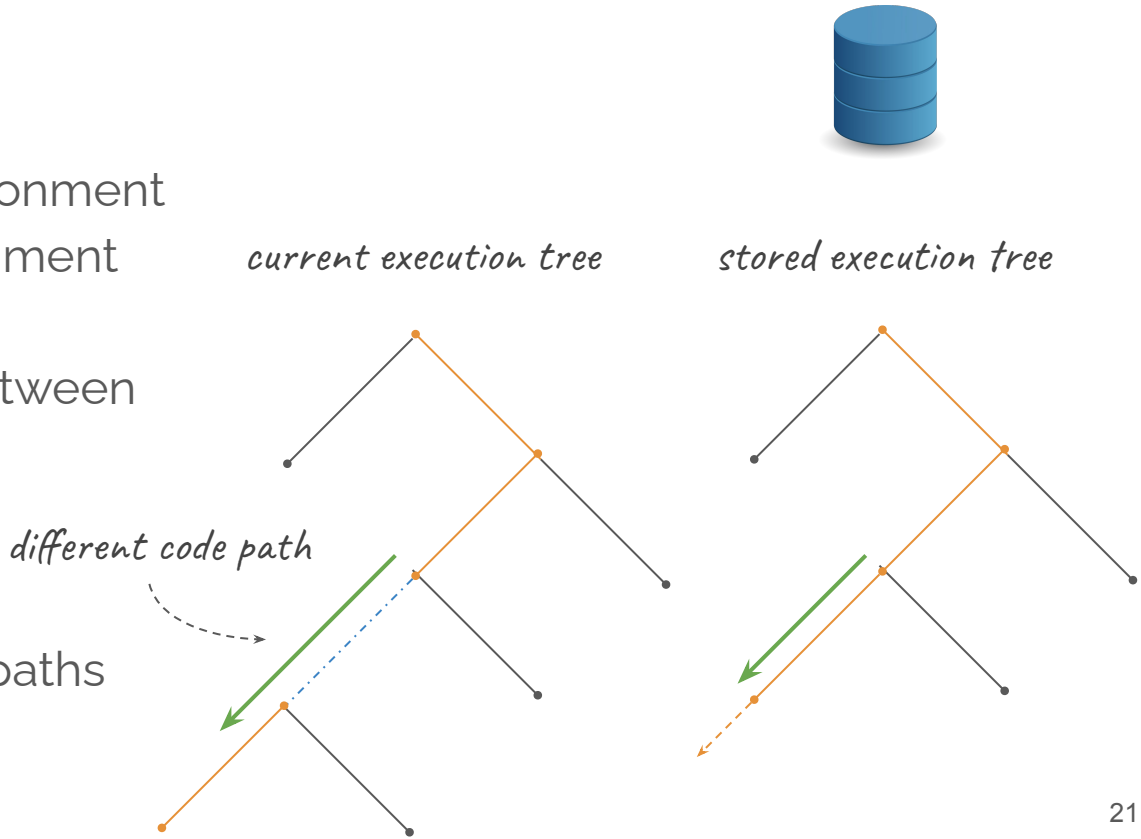
Divergences

Causes

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

Problem

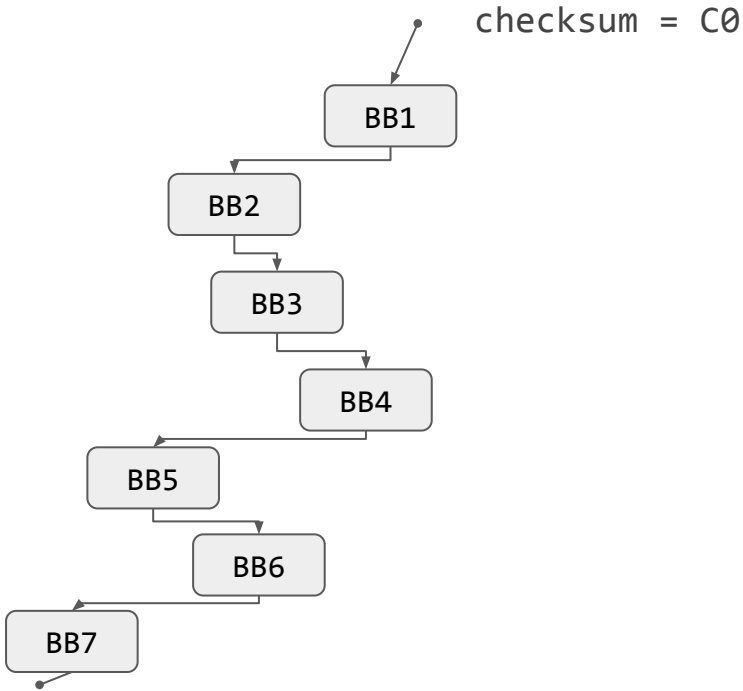
- exploration of infeasible paths
- false positives/negatives



Divergence Detection

Mitigation

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

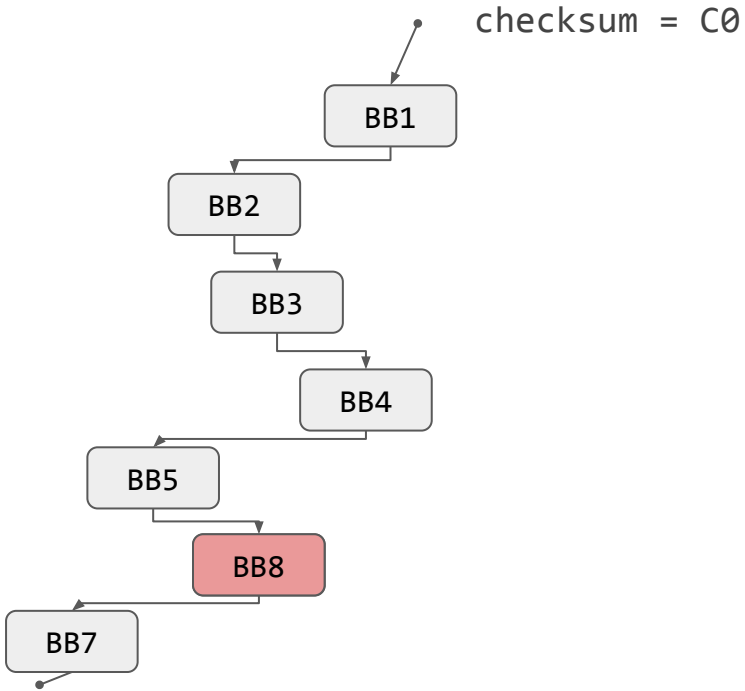


checksum = hash(BB7) ⊗ hash(BB6) ⊗ ... ⊗ C0

Divergence Detection

Mitigation

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

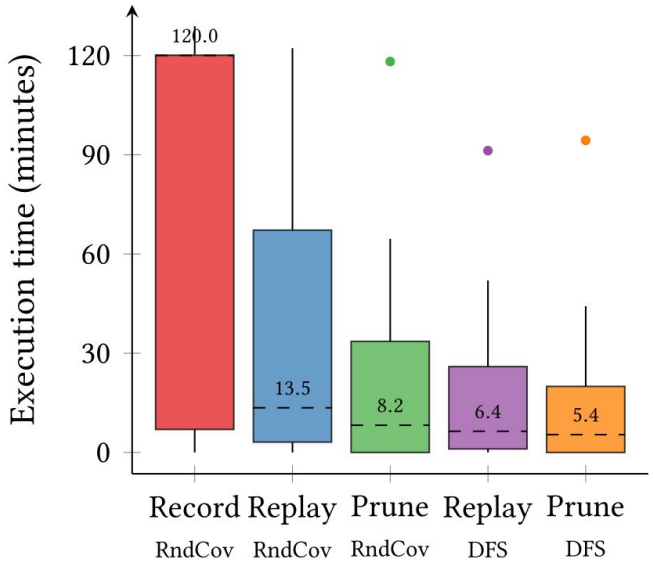


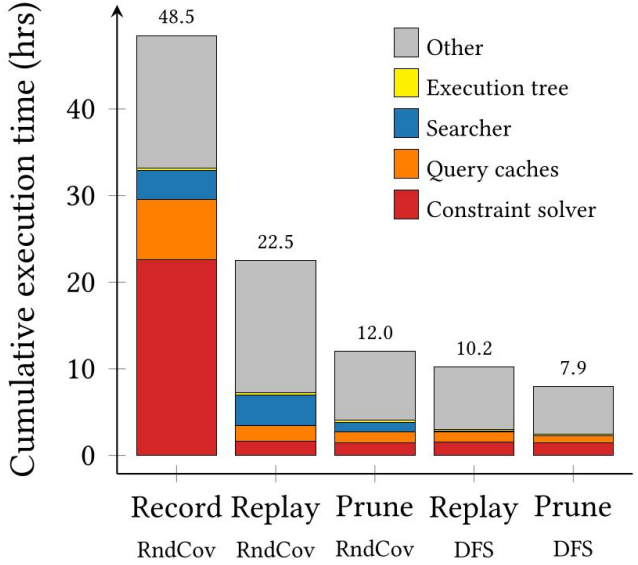
`checksum ≠ hash(BB7) ⊗ hash(BB8) ⊗ ... ⊗ C0`

Evaluation

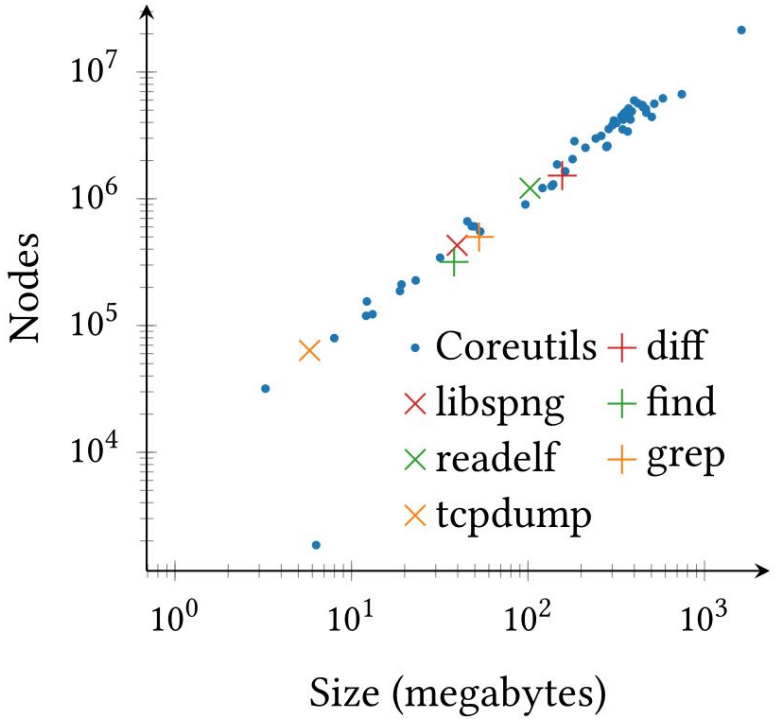
- MoKlee is implemented on top of KLEE 1.4
- evaluated on 93 benchmarks:
  - readelf (Binutils)
  - 87 Coreutils
  - diff (Diffutils)
  - find (Findutils)
  - grep
  - libspng
  - tcpdump

Evaluation - Runtime

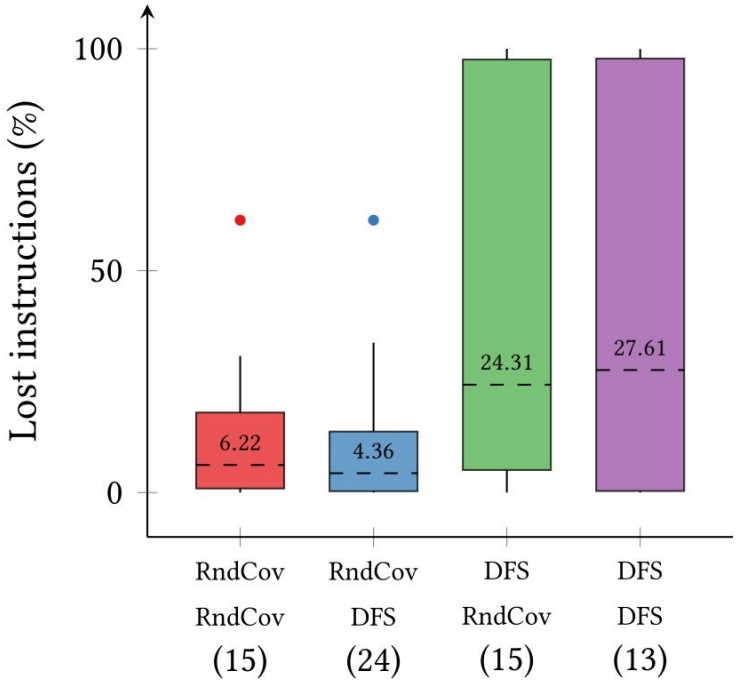




Evaluation - Storage Size



Evaluation - Divergences



Evaluation - Long Running Symbolic Execution

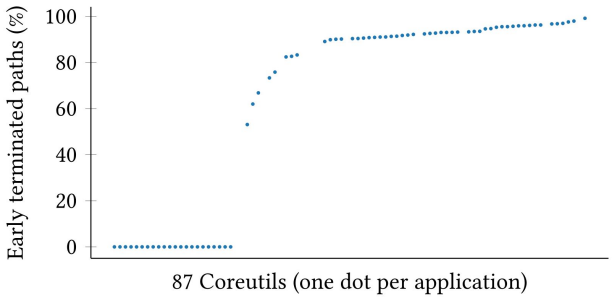
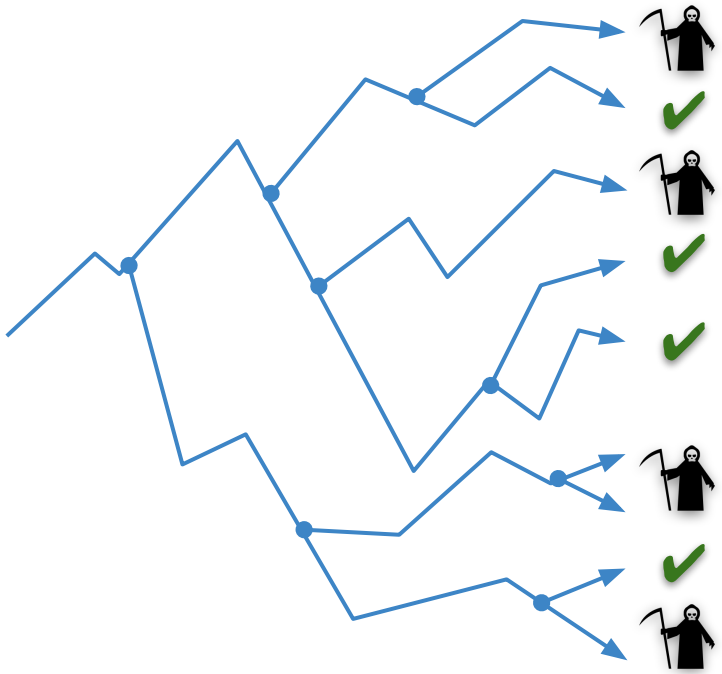
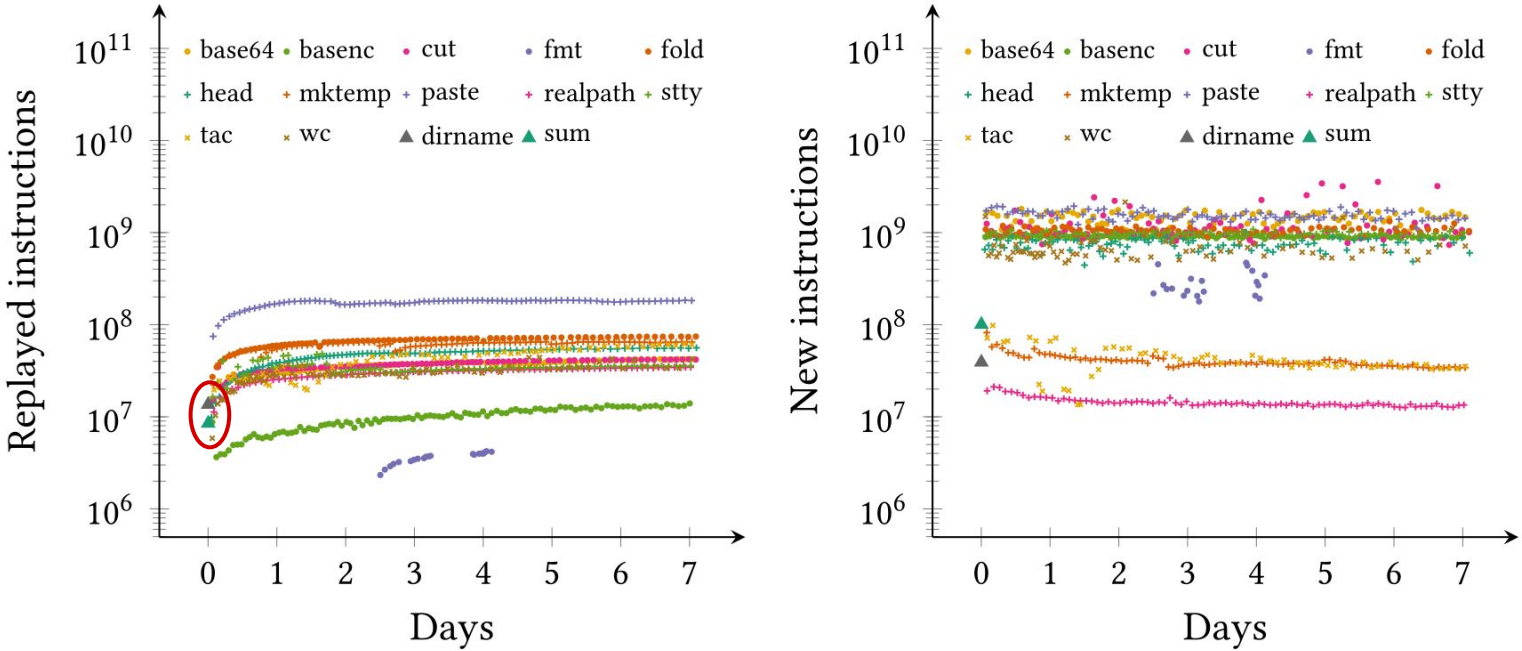


Figure 1: When running KLEE[1] on 87 *Coreutils* for 2 h each with the default search heuristic and memory limit (2 GB), most paths are terminated early due to memory pressure.

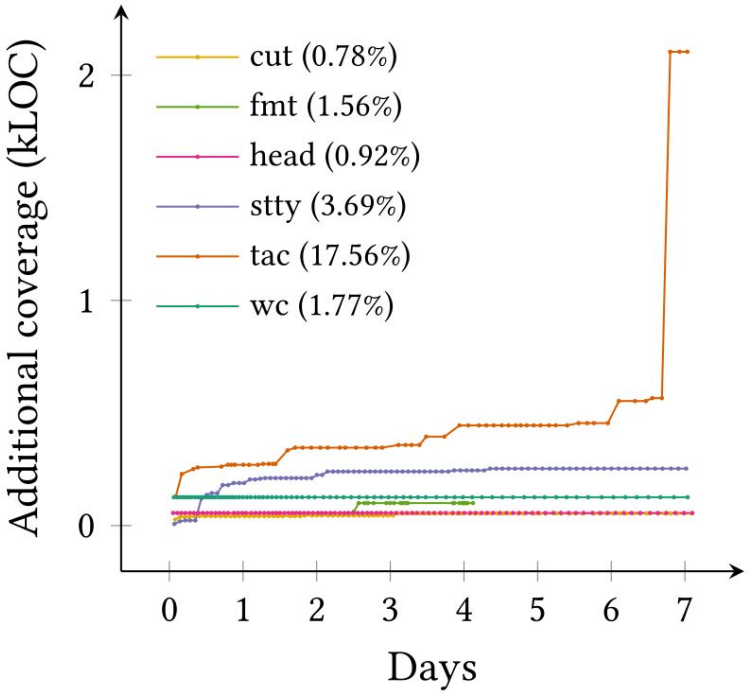
14 applications terminate states early and then run out of states before the 2h limit!

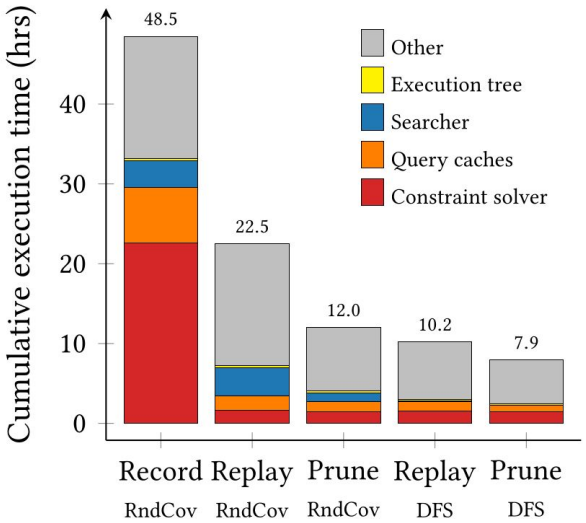


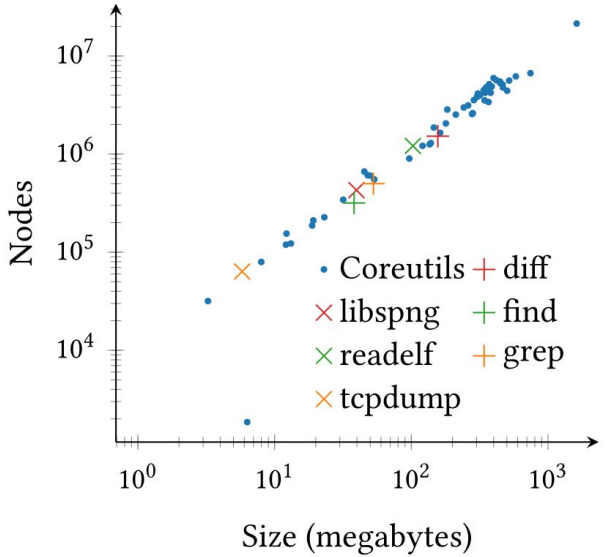
Evaluation - Long Running Symbolic Execution

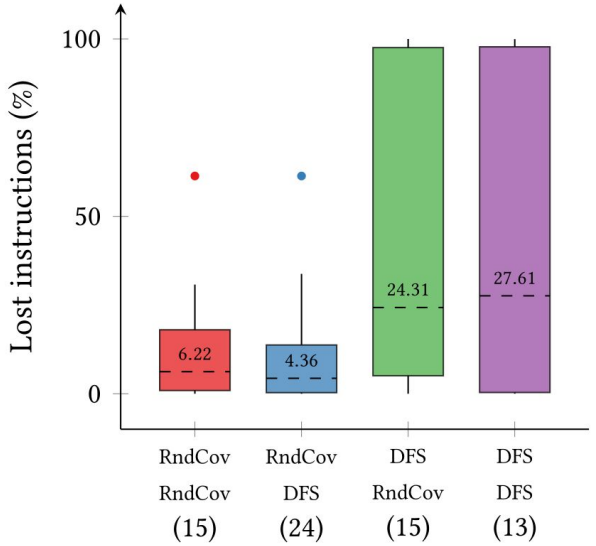


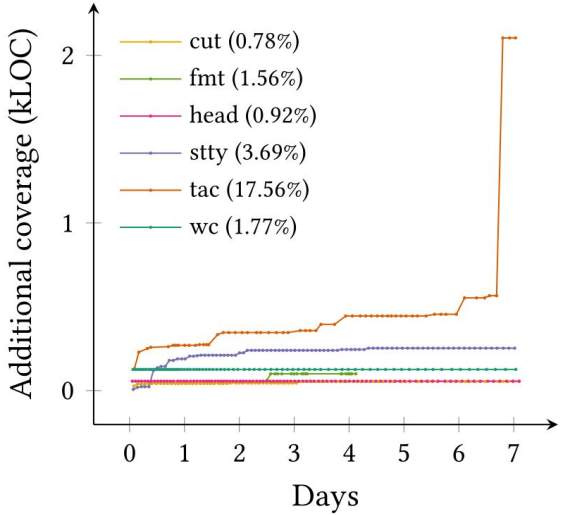
Evaluation - Long Running Symbolic Execution











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