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
Challenges in Symbolic Execution

- Constraint solving overhead
  - feasibility checks
  - safety checks
  - test generation

Path explosion
Early Termination (Memory Pressure)

![Graph showing early terminated paths (%) vs. 87 Coreutils (one dot per application)]
Motivation

don’t re-solve queries

large subtree

don’t re-explore paths
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

1. load metadata from database
2. re-use solver results
3. branch
Memoization

current execution tree

1. load metadata from database
2. re-use solver results

3. branch
4. load metadata
5. free metadata in parent

stored execution tree
Memoization

current execution tree

stored execution tree

path progresses beyond memoized data
Memoization

current execution tree

path switches to recording mode

stored execution tree
Memoization

current execution tree

stored execution tree

metadata is freed

metadata in database is updated
Memoization

current execution tree

stored execution tree

new subtree is written to database
Path Pruning

on branch completeness
immediately detected

current execution tree

completed subtree

stored execution tree
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
- ...

Database
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.
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.

(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>
<thead>
<tr>
<th>Depth</th>
<th>Sym Exe at Depth A</th>
<th>Sym Exe at Depth B</th>
<th>Sym Exe at Depth C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (ss)</td>
<td>Mem (MB)</td>
<td>States</td>
</tr>
<tr>
<td>A</td>
<td>Reg</td>
<td>ID</td>
<td>Reg</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td>29</td>
<td>30</td>
<td>34</td>
<td>60</td>
</tr>
</tbody>
</table>

(a) WBS Example

<table>
<thead>
<tr>
<th>Depth</th>
<th>Sym Exe at Depth A</th>
<th>Sym Exe at Depth B</th>
<th>Sym Exe at Depth C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (ss)</td>
<td>Mem (MB)</td>
<td>States</td>
</tr>
<tr>
<td>A</td>
<td>Reg</td>
<td>ID</td>
<td>Reg</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>35</td>
<td>38</td>
</tr>
<tr>
<td>29</td>
<td>30</td>
<td>86</td>
<td>87</td>
</tr>
<tr>
<td>34</td>
<td>35</td>
<td>96</td>
<td>97</td>
</tr>
</tbody>
</table>

(b) MerArbiter Example

<table>
<thead>
<tr>
<th>Depth</th>
<th>Sym Exe at Depth A</th>
<th>Sym Exe at Depth B</th>
<th>Sym Exe at Depth C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (ss)</td>
<td>Mem (MB)</td>
<td>States</td>
</tr>
<tr>
<td>A</td>
<td>Reg</td>
<td>ID</td>
<td>Reg</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>127</td>
<td>125</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>538</td>
<td>549</td>
</tr>
</tbody>
</table>

(c) Apollo Example

*(short runtimes)*
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

<table>
<thead>
<tr>
<th>Depth</th>
<th>Sym Exe at Depth A</th>
<th>Sym Exe at Depth B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (ss)</td>
<td>Mem (MB)</td>
</tr>
<tr>
<td></td>
<td>Reg</td>
<td>ID</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>16</td>
</tr>
<tr>
<td>29</td>
<td>30</td>
<td>34</td>
</tr>
</tbody>
</table>

(a) WBS Example

<table>
<thead>
<tr>
<th>Depth</th>
<th>Sym Exe at Depth A</th>
<th>Sym Exe at Depth B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (ss)</td>
<td>Mem (MB)</td>
</tr>
<tr>
<td></td>
<td>Reg</td>
<td>ID</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>35</td>
</tr>
<tr>
<td>29</td>
<td>30</td>
<td>86</td>
</tr>
<tr>
<td>34</td>
<td>35</td>
<td>96</td>
</tr>
</tbody>
</table>

(b) MerArbiter Example

<table>
<thead>
<tr>
<th>Depth</th>
<th>Sym Exe at Depth A</th>
<th>Sym Exe at Depth B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (ss)</td>
<td>Mem (MB)</td>
</tr>
<tr>
<td></td>
<td>Reg</td>
<td>ID</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>127</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>538</td>
</tr>
</tbody>
</table>

(c) Apollo Example

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

\[
\text{checksum} = \text{hash}(\text{BB7}) \odot \text{hash}(\text{BB6}) \odot \ldots \odot C0
\]
Divergence Detection

Mitigation

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

\[
\text{checksum} \neq \text{hash(BB7)} \otimes \text{hash(BB8)} \otimes \ldots \otimes C0
\]
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

![Runtime Evaluation Chart]

**Execution time (minutes)**

- **Record**:
  - RndCov: 120.0
  - RndCov: 13.5
- **Replay**:
  - RndCov: 8.2
  - DFS: 6.4
- **Prune**:
  - RndCov: 5.4

**Cumulative execution time (hrs)**

- **Record**:
  - RndCov: 48.5
  - RndCov: 22.5
- **Replay**:
  - RndCov: 12.0
- **Prune**:
  - DFS: 10.2
  - DFS: 7.9
Evaluation - Storage Size

![Graph showing nodes vs. size (megabytes)]
Evaluation - Divergences
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

Days

Additional coverage (kLOC)

- cut (0.78%)
- fmt (1.56%)
- head (0.92%)
- stty (3.69%)
- tac (17.56%)
- wc (1.77%)
MoKlee:
https://srg.doc.ic.ac.uk/projects/moklee/