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

\( \Omega \)

symbolic input

➔ 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

- don’t re-solve queries
- don’t re-explore paths

large subtree
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
4. load metadata
5. free metadata in parent
Memoization

Current execution tree

Path progresses beyond memoized data

Stored execution tree
Memoization

current execution tree

stored execution tree

path switches to recording mode
Memoization

current execution tree

stored execution tree

metadata in database is updated
Memoization

current execution tree

stored execution tree

metadata is freed
Memoization

current execution tree

stored execution tree

new subtree written to database
Path Pruning

current execution tree

stored execution tree

completed subtree
Path Pruning

on branch completeness
immediately detected

current execution tree

completed subtree

stored execution tree
Path Pruning

- *current execution tree*
- *stored execution tree*
- *completed subtree*

*path gets terminated*
Path Pruning

... and removed from tree

current execution tree

completed subtree

stored execution tree
Memoized Symbolic Execution

Guowei Yang
University of Texas at Austin
Austin, TX 78712, USA
guowei.yang@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 be brought back from the old trie.

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 long as divergence detection is disabled, all other values are preserved.

<table>
<thead>
<tr>
<th>Table 1: Iterative Deepening Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Depth</strong></td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>Reg</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>29</td>
</tr>
</tbody>
</table>

*a* WBS Example

<p>| Depth | <strong>Sym Exe at Depth A</strong> | <strong>Sym Exe at Depth B</strong> |
|-------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| A | B | Time (ss) | Mem (MB) | States | #Solver calls | Time (ss) | Mem (MB) | Tric (MB) |</p>
<table>
<thead>
<tr>
<th>Reg</th>
<th>ID</th>
<th>Reg</th>
<th>ID</th>
<th>Reg</th>
<th>ID</th>
<th>Reg</th>
<th>ID</th>
<th>Reg</th>
<th>ID</th>
<th>Reg</th>
<th>ID</th>
<th>Reg</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>25</td>
<td>35</td>
<td>38</td>
<td>304</td>
<td>376</td>
<td>17103</td>
<td>16756</td>
<td>12252</td>
<td>2942</td>
<td>47</td>
<td>46</td>
<td>45</td>
<td>413</td>
</tr>
<tr>
<td>29</td>
<td>30</td>
<td>86</td>
<td>87</td>
<td>419</td>
<td>333</td>
<td>33273</td>
<td>15250</td>
<td>25864</td>
<td>1540</td>
<td>92</td>
<td>45</td>
<td>45</td>
<td>413</td>
</tr>
<tr>
<td>34</td>
<td>35</td>
<td>96</td>
<td>97</td>
<td>419</td>
<td>345</td>
<td>35359</td>
<td>1476</td>
<td>27636</td>
<td>18</td>
<td>102</td>
<td>9</td>
<td>9</td>
<td>292</td>
</tr>
</tbody>
</table>

*b* MerArbiter Example

<p>| Depth | <strong>Sym Exe at Depth A</strong> | <strong>Sym Exe at Depth B</strong> |
|-------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| A | B | Time (ss) | Mem (MB) | States | #Solver calls | Time (ss) | Mem (MB) | Tric (MB) |</p>
<table>
<thead>
<tr>
<th>Reg</th>
<th>ID</th>
<th>Reg</th>
<th>ID</th>
<th>Reg</th>
<th>ID</th>
<th>Reg</th>
<th>ID</th>
<th>Reg</th>
<th>ID</th>
<th>Reg</th>
<th>ID</th>
<th>Reg</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>10</td>
<td>127</td>
<td>125</td>
<td>425</td>
<td>352</td>
<td>674</td>
<td>647</td>
<td>255</td>
<td>121</td>
<td>195</td>
<td>76</td>
<td>77</td>
<td>421</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>538</td>
<td>549</td>
<td>413</td>
<td>414</td>
<td>2243</td>
<td>2113</td>
<td>2160</td>
<td>966</td>
<td>1033</td>
<td>490</td>
<td>483</td>
<td>390</td>
</tr>
</tbody>
</table>

(c) Apollo Example

*No divergence detection.*

(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 = \text{hash}(\text{BB7}) \otimes \text{hash}(\text{BB6}) \otimes \ldots \otimes C0
Divergence Detection

Mitigation

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

Nodes

Size (megabytes)

10^{0}  10^{1}  10^{2}  10^{3}

10^{4}  10^{5}  10^{6}  10^{7}

- Coreutils + diff
- libspng  find
- readelf  grep
- tcpdump
Evaluation - Divergences

![Bar chart showing lost instructions (%) for different methods and iterations.]

- RndCov (15): 6.22
- RndCov (24): 4.36
- DFS (15): 24.31
- DFS (13): 27.61
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

![Graphs showing the evaluation of long running symbolic execution. The graphs display the number of read and new instructions over days for different programs and tools, such as `base64`, `basenc`, `cut`, `fmt`, `fold`, `head`, `mktemp`, `paste`, `realpath`, `stty`, `tac`, `wc`, `dirname`, and `sum`.](image-url)
Evaluation - Long Running Symbolic Execution

![Graph showing additional coverage over days](image)

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

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

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