Memoise: A Tool for Memoized Symbolic Execution

Guowei Yang and Sarfraz Khurshid
The University of Texas at Austin

Corina S. Păsăreanu
CMU Silicon Valley and NASA Ames

***Abstract*—This tool paper presents a tool for performing *memoized symbolic execution (Memoise)*, an approach we developed in previous work for more efficient application of symbolic execution. The key idea in Memoise is to allow re-use of symbolic execution results across different runs of symbolic execution without having to re-compute previously computed results as done in earlier approaches. Specifically, Memoise builds a *trie*-based data structure to record path exploration information during a run of symbolic execution, optimizes the trie for the next run, and re-uses the resulting trie during the next run. Our tool optimizes symbolic execution in three standard scenarios where it is commonly applied: *iterative deepening*, *regression analysis*, and *heuristic search*. Our tool Memoise builds on the Symbolic PathFinder framework to provide more efficient symbolic execution of Java programs and is available online for download. The tool demonstration video is available at http://www.youtube.com/watch?v=ppfYOB0Z2vY&feature=plcp.**

I. Introduction

Symbolic execution [4], [6] is a path-based program checking technique, which, in recent years, has been the focus of a number of research projects on automated input generation and bug finding [3], [5], [9], [10], as well as various other applications [7], [11]. The technique systematically explores the program paths (of interest) and for each path, builds a *path condition*, i.e., constraint (on program inputs) that represents the branching conditions on the path. The feasibility of these path conditions is checked using off-the-shelf constraint solvers as the conditions are encountered during symbolic execution to detect infeasible paths (if possible) and to enumerate test inputs that execute feasible paths. In principle, the technique can check rich functional correctness properties of code, e.g., with respect to given assertions. However in practice, the technique faces important challenges due to its inherent high computational complexity. There are two key factors that determine its cost: (1) the number of paths to explore; and (2) the cost of constraint solving.

In previous work [13] we introduced *memoized symbolic execution* (*Memoise*), an approach that addresses both these cost factors to enable more efficient applications of symbolic execution. Our key insight is that applying 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 leverages the similarities to reduce both the number of paths to explore by pruning the path exploration as well as the cost of constraint solving by re-using path feasibility results that were previously computed during constraint solving, thereby reducing the total cost of applying symbolic execution.

Memoise maintains and updates the *state* of a symbolic execution run using a *trie* [12] — an efficient tree-based data structure — which provides a compact representation of the symbolic paths visited during a symbolic execution run. Specifically, the trie records the *choices* taken when exploring different paths, together with bookkeeping information that maps each trie node to the corresponding condition in the code. 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. Constraint solving is turned off for previously explored paths and the search is guided by the choices recorded in the trie. Moreover, the search is pruned for the paths that are deemed to be no longer of interest for the analysis.

Memoise keeps its overhead low using two key operations on the trie: *compression* (to discard "un-interesting" trie branch sequences) and *merging* (of compressed tries constructed during successive runs of Memoise). Experimental results for three standard application scenarios for symbolic execution, namely iterative deepening, regression analysis, and heuristic search, validate the relatively small overhead for storing and retrieving the trie. Moreover, the experiments demonstrate the benefits and re-use Memoise enables.

This paper presents our tool embodiment of Memoise for Java programs. Memoise uses the Symbolic PathFinder tool [9], which is a part of the Java PathFinder (JPF) open-source framework. Memoise is publicly available for download at: https://hostdb.ece.utexas.edu/∼gyang/memoise.

To our knowledge, Memoise is the first tool to provide re-use of symbolic execution results across different runs of symbolic execution. Memoise opens a new exciting avenue for mitigating the path explosion problem that in inherent in symbolic execution. We believe the Memoise approach will substantially increase the practicality of using symbolic execution for larger code-bases.

II. Memoized Symbolic Execution

Given program p and execution depth bound b, *memoized symbolic execution* (*Memoise*) addresses the problem of running symbolic execution on *problem instance* $\langle p, b\rangle$ given that symbolic execution was already performed on problem instance $\langle p_{old}, b_{old}\rangle$. Memoise leverages the results of running

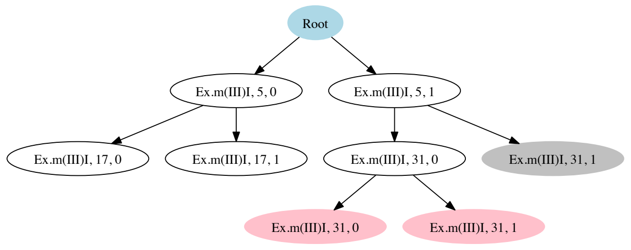

Fig. 1. Example trie

symbolic execution on $\langle p_{old}, b_{old} \rangle$ by caching them and re-using them when running symbolic execution on $\langle p, b \rangle$. To cache a symbolic execution run, Memoise builds an efficient *trie* data structure [12] for representing compactly the global state of a symbolic execution run, i.e. the choices taken during symbolic execution. The trie t is *complete* for program p and bound b if it encodes all the choices taken during the symbolic execution of p up to bound b. Memoise has three basic steps:

- **Initialization.** An initial run of Memoise performs standard symbolic execution as well as builds the trie on-the-fly and saves it on the disk for future re-use. Whenever a conditional instruction is symbolically executed a trie node is created, recording the location of the symbolic conditional, i.e., method and the instruction offset, and the choice taken by the execution. The trie stores just enough information to guide symbolic execution in future runs (through the stored choices) and to map back the nodes to the program constructs (for e.g. regression analysis). To facilitate future runs of symbolic execution, a subset of the leaf nodes in a trie is partitioned into a set of boundary nodes, which are leaf nodes because of the chosen depth bound, and a set of unsatisfiable nodes, which are leaf nodes due to unsatisfiable path conditions. Fig. 1 shows an example trie (for depth 4). In this figure, the two pink nodes are boundary node and the grey node is an unsatisfiable node.
- **Memoized analysis.** The trie built during the initialization run or a previous run of Memoise is loaded in memory and it is used to guide analysis for iterative deepening, regression analysis, or application of heuristics. During the analysis, a new trie is built/updated on-the-fly, which is saved back on the disk. As an optimization, the memoized analysis performs a *compression* on the input trie to remove the components that are irrelevant in the context of the particular application scenario.
- **Trie merging.** The (compressed) trie built during memoized analysis is (optionally) *merged* with the old trie to obtain a complete trie for $\langle p, b \rangle$.

A. Enabled Applications

Memoise can be used to optimize *many* applications [13]. We describe here three well-known applications of symbolic execution that are supported by our tool: symbolic execution with iterative deepening, regression analysis, and heuristic search to enhance program coverage. Other applications, such as continuous testing, load balancing for parallel execution, partial symbolic execution, component certification are described elsewhere [13].

For these applications, a key step is based on the trie and the particular application scenario to compute paths (of interest) that need re-executed in the new run of symbolic execution.

1) Iterative Deepening: In iterative deepening, the search depth is iteratively increased until either an error is found or the desired testing coverage has been achieved. For this particular analysis, only paths bounded by the search depth bound need to be re-explored during in the new iteration, since other paths are ended naturally by execution.

Therefore, Memoise enables an efficient iterative deepening approach by re-using the trie collected from smaller depths when exploring paths at larger depths. The approach works as follows. In one iteration, paths are explored exhaustively up to a certain depth and the choices made during the symbolic execution are stored in the trie structure. When the search depth bound is hit, the current trie node at that point is a boundary node. The paths that lead to boundary nodes are then selected and, guided by the trie, are executed up to the next depth bound. Note that other paths would not be re-executed (e.g. the paths who ended at smaller depths can not have successors at the new bigger depth). During re-execution we turn off constraint solving for the portion of the path that has been already explored in the previous iteration, and the exploration is only guided by the choices recorded in the trie. The process repeats until all paths are explored, or the new bound is reached.

2) Regression Analysis: Programs evolve during development or maintenance. Reapplying full symbolic execution to programs as they evolve may be impractical. In regression analysis, program differences are utilized to make symbolic execution more efficient on the subsequent program version.

Memoise enables regression analysis by only allowing the paths impacted by the program change to be re-executed. A change impact analysis is used to identify the *impacted* trie nodes, which represent roots of sub-trees potentially changed by the execution of the change during memoized execution. Thus, only paths leading to the impacted trie nodes need to be re-executed in this particular analysis. As before, for the portion of the path up to the impacted node, constraint solving is turned off, and only the part rooted at the impacted node needs to be rebuilt while it is explored using constraint solving.

3) Heuristics-Guided Symbolic Execution: The iterative-deepening approach described above can be further extended to perform a heuristic search of program paths, as guided by the testing coverage achieved so far. At each iteration, the approach discovers those paths that may lead to increased code coverage, and selects only those paths for re-execution up to larger depths in subsequent iterations.

TABLE I
ITERATIVE DEEPENING RESULTS

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

III. IMPLEMENTATION AND RESULTS

A. Tool Implementation

Memoise is built on top of Symbolic PathFinder (SPF) [8], [9], an open source symbolic execution tool for Java bytecode. SPF is part of the Java PathFinder verification tool-set [1] which includes an explicit-state software model checker, and several extension projects, one of them being SPF.

The procedures for building the trie, iterative deepening, and regression analysis are implemented as JPF *listeners*, which monitor the execution of the program inside SPF. When building the trie, JPF's search events such as "state advanced" and "state backtracked" are monitored, so that whenever a conditional instruction is symbolically executed a trie node is created as a child of the current trie node, and the current trie node is updated while the search is advanced or backtracked correspondingly. Information including the conditional instruction bytecode offset, the choice taken by execution, and the fully qualified method name, is collected at runtime and stored in the trie. When the search depth bound is hit, the current trie node at that point is marked as *boundary*. The saving/loading of tries is implemented using the Java Serialization API, which stores Object state to a file in disk.

B. Evaluation

We have performed experiments to evaluate Memoise for several analyses, including symbolic execution with iterative deepening, regression analysis and guidance heuristics [13]. We show here briefly some results for one of the analysis, namely symbolic execution with iterative deepening.

Table I shows the results of applying Memoise with iterative deepening for MerArbiter, a component of the flight software for NASA JPL's Mars Exploration Rovers (MER). MerArbiter has been modeled in Simulink/Stateflow and it was automatically translated into Java using the Polyglot framework [2]. The example has 268 classes, 553 methods, 4697 lines of code (including Polyglot). We show the results of increasing the depth from A to B for three experiments. At depth A we built the trie while at depth B, we re-used and updated the trie. We also conducted regular symbolic execution as implemented in SPF at both depth A and depth B. Table I shows the time and memory results for regular symbolic execution and for Memoise. It also shows the number of states, the number of constraint solver calls and the size of Trie that is saved during Memoise at depth B. Reg represents regular symbolic execution while Memoise represents Memoise for iterative deepening. Mem-c and Mem-p respectively represent Memoise with compression and without compression. For symbolic execution at depth A, we find that the time cost of Memoise is is similar to regular symbolic execution. For symbolic execution at depth B, Memoise typically explores fewer states, makes fewer solver calls, and correspondingly takes less time. In *MerArbiter* when the depth is increased from 34 to 35, the reduction is more than an order of magnitude. Moreover, the reduction for the number of states, solver calls, and time appears to get more significant when the depth goes deeper. The table also shows that compression makes the trie smaller.

IV. EXAMPLE

In this section, we use a small example program, `Ex` as included in the example package of the `jpf-memoise` repository, to illustrate how to use Memoise for iterative deepening and regression analysis. We also describe how Memoise supports heuristic search.

A. Initialization

An initialization run of Memoise is needed to build a trie on-the-fly and store it on the disk for re-use. The following code shows an example `.jpf` configuration to do so:

```
1 target=Ex
2 classpath=${jpf-memoise}/build/examples
3 symbolic.method=Ex.m(sym#sym#sym)
4 search.depth_limit=6
5 listener=gov.nasa.jpf.memoise.listener.TrieBuilder
6 memoise.new_trie_name=trie_ex.dat
```

The first four lines in the configuration specify the target class, its class path, the method to execute symbolically, and search depth bound; these lines are similar to writing a standard `*.jpf` configuration to run regular symbolic execution with Symbolic PathFinder. Line 5 specifies the TrieBuilder listener which serves to build a trie during memoized symbolic execution, and line 6 specifies the name of the trie.

After running Memoise, a trie file is generated, which can be printed to a dot graph for visualization using the util class provided. The generated trie is shown as Fig. 1. The gray node is an unsatisfiable node due to the unsatisfiable path condition, and the two pink nodes are boundary nodes because of the chosen depth bound.

B. Iterative Deepening

Suppose the user would like to increase the depth bound to check more program behaviors. The following code shows a `.jpf` configuration for Memoise:

```
1 target=Ex
2 classpath=${jpf-memoise}/build/examples
3 search.depth_limit=7
4 symbolic.method=Ex.m(sym#sym#sym)
5 listener=gov.nasa.jpf.memoise.listener.IDListener
6 memoise.old_trie_name=trie_ex.dat
7 memoise.new_trie_name=trie_ex_new.dat
```

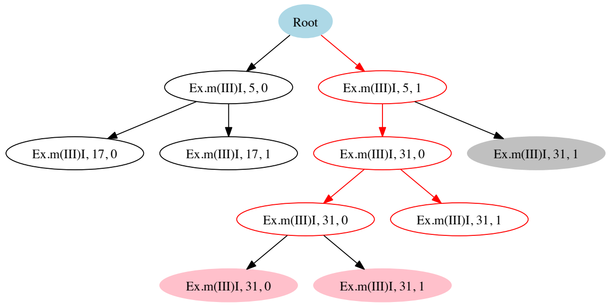

Fig. 2. Updated trie in iterative deepening

The depth bound is increased from 6 to 7, the `IDListener` is applied for iterative deepening using Memoise. Memoise uses the old trie to guide symbolic execution to explore the new paths because of the increased depth bound, and updates the trie accordingly. The last two lines specify the name of the trie for reuse and the name of the updated trie, respectively.

After running Memoise, an updated trie shown in Fig. 2 is generated. The trie shows that only nodes (paths) highlighted in red color are re-explored, while the others that also exist in the old trie are pruned during search. The two boundary nodes (shown in pink color) are new, which are generated because of the increased depth bound.

C. Regression Analysis

Suppose the user would like to change the program to a new version, say due to a bug fix, and re-run symbolic execution to check the new program version. The two versions of our example program `Ex` are available in `jpf-memoise` repository. The following code shows a `.jpf` configuration for Memoise to do regression analysis:

```
1 target=Ex
2 classpath=${jpf-memoise}/version1
3 search.depth_limit=6
4 symbolic.method=Ex.m(sym#sym#sym)
5 listener=gov.nasa.jpf.memoise.listener.RSEListener
6 memoise.newClass=${jpf-memoise}/version1/Ex.class
7 memoise.oldClass=${jpf-memoise}/version0/Ex.class
8 memoise.old_trie_name=trie_ex.dat
9 memoise.new_trie_name=trie_ex_new.dat
```

Line 5 specifies that *RSEListener* is applied, and lines 6 and 7 specify the locations of the two versions of the `EX` class. Memoise compares the two class files to compute the change in control flow graphs, and map this information to the trie to find impacted trie nodes. Only paths leading to those impacted trie nodes are re-explored during symbolic exeuction, whereas the other paths can be pruned from the search. After running Memoise, an updated trie (Fig. 3) is generated. We find that only one path, which is highlighted in red color is re-explored. Because the change is made in the second executed branch (choice 1) of this particular conditional (bytecode 17), the highlighted path is the only path that could be potentially impacted by the change. Since the change does not introduce any new path, the updated trie remains the same as the before.

Our tool Memoise can also be used in the context of heuristic search using the following options:

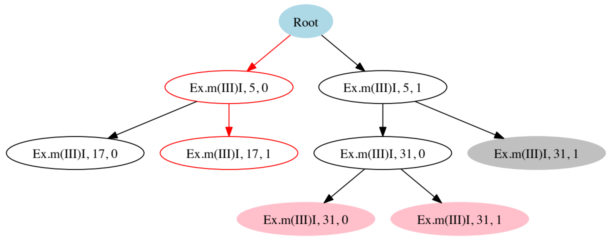

Fig. 3. Updated trie in regression analysis

- HRListener_CT, which supports Counter heuristic that favors paths with the maximum number of specific branches;
- HRListener_RC_M, which supports Reachability heuristic that favors paths that end in certain method;
- HRListener_RC_C, which supports Reachability heuristic that favors paths that end in certain class;

V. Conclusions

We presented Memoise, a tool for memoized symbolic execution for Java programs. Memoise re-uses symbolic execution results across different runs of symbolic execution without having to re-compute previously computed results as done in earlier approaches. Memoise's current implementation supports symbolic execution in three standard scenarios where it is commonly applied: iterative deepening, regression analysis, and heuristic search. It reduces the analysis cost by using the trie to quickly guide the search for previously explored paths (with the constraint solving turned off) and by pruning the paths that are not relevant for the current run. Our tool Memoise builds on the Symbolic PathFinder framework and is publicly available for download.

References

[1] Java PathFinder Tool-set. http://babelfish.arc.nasa.gov/trac/jpf.

[2] D. Balasubramanian, C. S. Pasareanu, M. W. Whalen, G. Karsai, and M. R. Lowry. Polyglot: modeling and analysis for multiple statechart formalisms. In *ISSTA*, pages 45–55, 2011.

[3] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[4] L. A. Clarke. A program testing system. In *Proc. of the 1976 annual conference*, ACM '76, pages 488–491, 1976.

[5] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI*, pages 213–223, 2005.

[6] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[7] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *PLDI*, pages 504–515, 2011.

[8] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *ISSTA*, pages 15–25, 2008.

[9] C. S. Păsăreanu and N. Rungta. Symbolic PathFinder: symbolic execution of Java bytecode. In *ASE*, pages 179–180, 2010.

[10] K. Sen and G. Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *CAV*, pages 419–423, 2006.

[11] S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke. Combining symbolic execution with model checking to verify parallel numerical programs. *ACM Trans. Softw. Eng. Methodol.*, 17(2), 2008.

[12] D. E. Willard. New trie data structures which support very fast search operations. *J. Comput. Syst. Sci.*, 28:379–394, July 1984.

[13] G. Yang, C. S. Păsăreanu, and S. Khurshid. Memoized symbolic execution. In *ISSTA 2012*, pages 144–154, 2012.