Compositional Symbolic Execution with Memoized Replay

Rui Qiu*, Guowei Yang†, Corina S. Păsăreanu‡ and Sarfraz Khurshid*

*University of Texas, USA, {ruiqiu, khurshid}@utexas.edu
†Texas State University, USA, gyang@txstate.edu
‡CMU/NASA Ames, USA, corina.s.pasareanu@nasa.gov

Abstract—Symbolic execution is a powerful, systematic analysis technique that has received much visibility in the last decade. Scalability however remains a major challenge for symbolic execution. Compositional analysis is a well-known general purpose methodology for increasing scalability. This paper introduces a new approach for compositional symbolic execution. Our key insight is that we can summarize each analyzed method as a memoization tree that captures the crucial elements of symbolic execution, and leverage these memoization trees to efficiently replay the symbolic execution of the corresponding methods with respect to their calling contexts. Memoization trees offer a natural way to compose in the presence of heap operations, which cannot be dealt with by previous work that uses logical formulas as summaries for compositional symbolic execution. Our approach also enables efficient target oriented symbolic execution for error detection or program coverage. Initial experimental evaluation based on a prototype implementation in Symbolic PathFinder shows that our approach can be up to an order of magnitude faster than traditional non-compositional symbolic execution.

1. INTRODUCTION

Symbolic execution is a powerful, systematic program analysis technique that has found many applications in recent years, ranging from automated test case generation and error detection, to regression analysis, security analysis, continuous testing and program repair [4], [18], [22]. The technique enumerates the program paths (up to a given bound) and records the conditions on the inputs to follow the different paths, as dictated by the branches in the code. Off-the-shelf constraint solvers [1] are used to check the satisfiability of path conditions to discard those paths that are found to be infeasible. In practice, scalability is a major challenge in symbolic execution due to high computational demand.

Compositional analysis is a well-known general purpose methodology that has been used with success to scale up static analysis and software verification [6], [7], [9], [13], including symbolic execution [3], [10], [12]. The main idea is to analyze each elementary unit (i.e., a method or a procedure) in the program separately, and to store the analysis results in a summary (for that method or procedure), encoding the input-output behavior of the unit. Whole-program analysis results are then obtained by incrementally composing and utilizing the previously built summaries.

This paper introduces a new approach for compositional symbolic execution. Our key insight is that we can summarize each analyzed method as a memoization tree that captures the crucial elements of symbolic execution, i.e. the choices made along each path and the input path conditions (including constraints on the program’s heap) for complete paths. The memoization tree succinctly summarizes the feasible paths through the method and it does not explicitly encode the method’s outputs as is typically done in previous approaches. Instead, we define a composition operation that uses the memoization trees, in a bottom-up fashion, for efficient replay of symbolic execution of the methods in different calling contexts. During composition, constraint solving is only used at a method call site to determine which paths in the method summary are still feasible; these paths are then explored without any further constraint solver calls and the search is guided by the choices recorded in the memoization tree. This results in significant savings in analysis time due to reduced number of solver calls, as compared to non-compositional symbolic execution.

A key advantage of using the memoization trees is that they offer a natural way of handling the heap, which cannot be dealt with by previous work that uses logical formulas as summaries for compositional symbolic execution [3], [10], [12]. When composing a method summary with the actual calling context, we first perform a partial check of the heap constraints on the (possibly symbolic) heap of the calling context and then re-execute the method guided by the memoization tree, which naturally reconstructs the heap, and re-computes the outputs of the method.

We enhance our approach for two common application scenarios of symbolic execution: error-detection and test generation to achieve a certain coverage. We introduce two heuristics that specifically target reachability of program statements and states of interest, and thus facilitate error detection. Specifically, we explicitly mark memoization tree nodes along paths that lead to a target, e.g., an assert statement that may lead to an assertion violation. The heuristics use this information to prioritize and prune the exploration of symbolic paths in a bottom-up incremental way, with the goal of quickly propagating up the information about the specific target.

While our focus in this paper is on improving scalability of symbolic execution using our compositional approach, it offers a number of other potential benefits. For example, our approach enables more efficient selective regression testing, where the memoization trees are stored off-line and re-used, e.g., when the code in the caller method is modified but the callee is unmodified. Moreover, our approach lends itself
naturally to parallel analysis for both building and re-use of summaries; method summaries for different methods can be constructed in parallel and memoized path conditions can be checked for the current calling context in parallel.

We implemented our approach into a prototype tool that builds on Symbolic PathFinder (SPF) [19]. The experimental results on 10 subjects show that compared to traditional non-compositional symbolic execution, our approach is up to an order of magnitude faster and has up to 88% reduction in constraint solver calls.

II. BACKGROUND

Symbolic execution [8], [16] is a technique that analyzes a program using symbolic values for inputs rather than actual concrete inputs as normal execution of the program would. In symbolic execution, program variables and outputs are computed as expressions in terms of those symbols from inputs. To determine what inputs lead to which paths of the program to be executed, symbolic execution introduces path constraints (PC) that are boolean expressions in terms of input symbols for possible choice of branch conditions in program. A symbolic execution tree represents the paths taken in a program during symbolic execution. Each node in the tree represents a state of the symbolic execution, which reflects a set of states in actual concrete execution. Edges between nodes stand for transitions among states.

We illustrate symbolic execution on a program in Figure 1 that has two methods p and q. Method p takes two integers x and y as input and returns an integer according to the relationship between x and y. Method q also takes two integers a and b as input and invokes method p to return an integer. We treat method q as the start point of symbolic execution. Figure 2 shows the complete symbolic execution tree of method q. Initially, PC is true, and symbolic variables a and b have symbolic values A and B respectively. Program variables are then set symbolic values in terms of A and B. For example, when method p is invoked in line 3 in method q, the values for input of method p (x and y) are A + 1 and B - 10 respectively. For each conditional statement in the program, PC will be updated with all possible choices from the branch condition so that all possible paths are explored. Whenever PC is updated, it is checked for satisfiability by calling the off-the-shelf constraint solver. If PC becomes false (not satisfiable), which means the corresponding path is infeasible, symbolic execution does not continue for that path. For example, in method q there are four paths that are infeasible due to the unsatisfiable path conditions. Programs with loops may have infinite numbers of paths so symbolic execution needs to be bounded for these programs. The exploration of paths can stop when a certain search depth is reached or a coverage criteria has been achieved.

Symbolic PathFinder (SPF) uses lazy initialization [15] to handle dynamic input data structures (e.g., lists and trees). The components of the program inputs are initialized on an "as-needed" basis. The intuition is as follows. To symbolically execute method m of class C, SPF creates a new object o of class C, leaving all its fields uninitialized. When a reference field e of type T is accessed in m for the first time, SPF nondeterministically sets e to null, a new object of type T with uninitialized fields, or an alias to a previously initialized object of type T. This enables the systematic exploration of different heap configurations during symbolic execution.

III. EXAMPLE COMPOSITIONAL ANALYSIS

This section illustrates our approach to compositional symbolic execution on the simple example from Figure 1.

When traditional (non-compositional) symbolic execution is applied on the method q shown in Figure 1, method p is executed twice, since both branches of the conditional statement at line 2 of method q are feasible. The cost of method p’s “re-execution” can be reduced by compositional symbolic execution, where we first build a memoization tree of method p, and then efficiently perform symbolic execution of method q by replaying the symbolic execution of p in the two calling contexts using p’s memoization tree.

Figure 3 (a) shows the memoization tree for method p. A memoization tree succinctly summarizes all the choices taken during symbolic execution [25]. Other than the root node n0, each node is created whenever a conditional statement is executed, recording the branch that is taken during symbolic execution, e.g., node n1 in Figure 3 (a) indicates that the true (1) branch of the conditional statement at line 2 in program p is executed. Additionally, the tree leaves are annotated with the path conditions for each complete path through the method. Out of the four paths in the program, three paths are captured in this memoization tree, because the missing path is infeasible with an unsatisfiable path condition: x < y && x == y + 1.

To replay the symbolic execution of the callee method that has a memoization tree, the (feasible) paths in the memoization
tree are checked against the calling context to determine whether they are feasible or not. We map the actual inputs of the callee method to the parameters stored in memoization tree. For example, when invoked at line 3 of method q, method p’s actual inputs are a + 1 and b – 10, and its formal arguments are x and y. So we map a + 1 → x and b – 10 → y. Each annotated path condition in the memoization tree is transformed by replacing the parameters with the actual inputs using the map, and then combined with the path condition from the calling context. The combined constraints are then checked to decide if the corresponding path in p is feasible or not. For the path in method p that ends at node n5 in Figure 3 (a), for instance, the transformed path condition (a + 1)! = (b – 10) + 1 & (a + 1) <= (b – 10) after replacing the formal arguments, combined with the calling context constraint a > b, is not satisfiable and thus this path is not feasible in the calling context. In this case, the nodes along this path, n2 and n5, are marked so that the path will not be explored by symbolic execution in composition.

Figure 3 (b) shows the memoization tree built for method q by reusing method p’s memoization tree. For paths in p that are feasible in the calling context, we do not put their corresponding nodes in q’s memoization tree. Instead, we use summary nodes to point to those paths in method p’s memoization tree to reduce the memory cost. For example, nodes n4 and n5 in Figure 3 (b) are summary nodes. In node n4, “[p : n4]” points to the path ended at node n4 (n0 → n1 → n4) in method p’s memoization tree.

In traditional symbolic execution, whenever the path condition is updated it is checked for satisfiability using the underlying constraint solver. Therefore, traditional symbolic execution makes 10 constraint solver calls in total for symbolically executing method q (according to the conditions encoded in Figure 2). In contrast, our approach makes 8 constraint solver calls, i.e., two calls for branch conditions in method q and 3 calls each in method p in its two calling contexts (line 3 and line 5 in method q), reducing the overall analysis time.

Although in this particular example the saving for number of constraint solver calls is rather modest, the benefits of the compositional analysis become more pronounced when there are many invocations of the same method, e.g., q is invoked in a loop, and each invocation only costs 2 and 3 constraint solver calls for methods q and p respectively (see Section V for benefits).

IV. COMPOSITIONAL SYMBOLIC EXECUTION

Overview. In our compositional approach the methods of a program are processed in an order corresponding to a bottom-up traversal of the program’s call graph, starting with the ones that invoke no other methods and incrementally processing methods whose sub-methods have already been processed until the whole program is analyzed. For each processed method, we use bounded symbolic execution to compute a method summary, which consists of a tree that succinctly represents all the symbolic paths through the method, together with the input path constraints for the complete paths; the bound is specified by the user and it is stored in the tree. The summary is stored (“memoized”) for future re-use, whenever that method is invoked from another method – we say that the two methods are composed.

Note that, when deciding on the order to analyze the methods in a given system, two main strategies can be followed. A top-down approach may be adequate if one wants to compute only the strictly necessary information. However, this approach does not guarantee that reuse of summaries is always possible, and summaries may need to be recomputed for different calling contexts. In contrast, a bottom-up approach ensures that the computed summaries can always be reused, albeit at the price of computing summaries larger than necessary in some cases. We follow the latter strategy in our framework.

At a high level, our approach works as follows. Let us assume that program P consists of an entry method M0 and a set of methods M1, M2, ..., Mk that belong to M0’s method invocation hierarchy. In the hierarchy, each leaf method, i.e., a method that invokes no other methods, is analyzed first, and its feasible paths and corresponding path constraints are summarized in a memoization tree. Then the methods that directly invoke these leaf methods are analyzed by leveraging the already built memoization trees. The process continues in a bottom-up fashion until the entry method M0 is analyzed.

Handling recursion. If a method invokes itself, we build the summary tree directly up to a pre-specified bound. For indirect recursion, i.e., a loop consisting of several methods in the call graph, we randomly choose one method in the loop and build the summary tree for it, without reusing other callee methods. The summary built will consist of all methods in the loop up to a certain bound.

Algorithm. Algorithm 1 describes our overall approach. Procedure getStaticCallGraph(P, M0) (Line 2) creates the call graph for P rooted in M0. Procedure getLeafNodes (Line 3) identifies the leaf methods in the invocation hierarchy. Furthermore, the algorithm checks whether each analyzed method has conditional statements using helper method containsCondStmt (Line 6), and skips the analysis of the method that does not contain any conditional statements, since methods with no conditional statements would not give benefits during compositional symbolic execution.

Procedure analyzeMethod (Line 7) performs a bounded symbolic execution for method m and builds its memoization tree using the previously computed memoization trees stored.
Algorithm 1 Compositional symbolic execution

Input: Program $P$, Entry method $M_0$
Output: A set of summary trees $S_T$ for methods in $P$
1: $S_T ← ∅$
2: CallGraph $cg ← getStaticCallGraph(P, M_0)$
3: Set $s ← cg.getLeafNodes()$
4: while $s ≠ ∅$ do
5:   for each Method $m ∈ s$ do
6:     if $m$.containsCondStmt() then
7:       SummaryTree $t ← analyzeMethod(m, S_T)$
8:       $S_T.add(t)$
9:   $s ← cg.getNodesToProcess(s)$
10: return $S_T$

in the set $S_T$. Procedure $getNodesToProcess$ (Line 9) returns the nodes in the call graph whose corresponding methods have not been processed so far but the methods that they invoke have already been processed.

Algorithm 2 shows procedure $analyzeMethod$ for building (Section IV-A) and composing (Section IV-B) method summary trees. We describe some of its key elements in more details below.

A. Construction of Memoization Trees

A memoization (or summary) tree is a recursive tree data structure that succinctly captures the crucial elements of symbolic execution for each analyzed method.

Each tree contains two types of nodes: normal nodes and summary nodes. At a high level, for each method $M_{callee}$ invoking method $M_{caller}$, normal nodes encode the choices taken for each condition in the code, while summary nodes encode pointers to the paths in $M_{callee}$’s summary that are found to be feasible at the invocation point from $M_{caller}$.

Normal Nodes. A “normal” node, $N[m: offset: choice]$, represents a choice taken at a conditional statement and it encodes: $m$ – the name of the enclosing method, $offset$ – the instruction offset of the conditional statement, and $choice$ – choice taken by the execution ("1" for true branch and "0" for false branch). For example, in Figure 3 (a), node $n2$ has a tuple $[p:2:0]$, indicating that in method $p$ instruction with offset $2$ ("$i f(x > y)$") takes the choice of false branch.

A symbolic execution path can be succinctly represented by the sequence of choices taken during its execution, and can be recovered from the memoization tree by traversing the tree from the root to a leaf. We thus can use the leaf nodes to represent their corresponding paths. For example, in Figure 3 (a), node $n4$ implies an execution path that takes the true branch at the first conditional statement and the false branch in the second conditional statement in method $p$.

We note that we need to keep track of all the conditions in the method, not just the ones being executed symbolically. The reason is that during replay, some of the conditions that were symbolic during tree generation may become concrete due to concrete inputs from the calling context; we cannot distinguish that from a condition that was concrete to begin with in the tree. We therefore chose to record all the conditions in the summary tree and used the tree to guide the execution of all the conditions (see lines 8-10 in $analyzeMethod$).

Algorithm 2 Procedure $analyzeMethod$ for building and composing memoization trees

Input: Method $M_{callee}$, Set of $MemoizationTree$ $S_T$
Output: MemoizationTree $T_{callee}$ for method $M_{callee}$
1: boolean $toCompose ← false$
2: MemoizationTree $T_{callee} ← new MemoizationTree()$
3: MemoizationTree $t ← null$
4: Instruction $insnToExe ← getNextInstruction()$
5: while $insnToExe ! = null$ do
6:   if $toCompose$ then
7:     $turnOnConstraintSolver()$
8:   if $type(insnToExe) == ConditionalInstruction then
9:     Node $n ← createNode(insnToExe)$
10:    $T_{callee}.add(n)$
11:   if $type(insnToExe) == InvokeInstruction then
12:     if invokedMethod(insnToExe) $∈ S_T$ then
13:       $toCompose ← true$
14:      $t ← S_T.getSummary(insnToExe)$
15:     Mapping $mapping ← createMapping()$
16:     Context context $← getSymExContext()$
17:     for each Path $path ∈ t.getPaths()$ do
18:       SummaryPC $spc ← path.getSummaryPC()$
19:       boolean $isConsistent ← false$
20:     for each $PCPair pcp ∈ spc$ do
21:       if $check(pcp, context, mapping)$ then
22:         $isConsistent ← true$
23:         break
24:     if $isConsistent$ then
25:       markNodes(path)
26:     else
27:       $turnOffConstraintSolver()$
28:     if $type(insnToExe) == ReturnInstruction then
29:       $toBackCaller(insnToExe)$ then
30:       $toCompose ← false$
31:     $T_{callee}.compressNode()$
32:   else
33:     $type(insnToExe) == ConditionalInstruction then
34:     Node $n ← t.getNextNode()$
35:   if $n ∈ SummaryNode$ then
36:     $t.decompressNode(n)$
37:   if $n ← getNextNode()$
38:   if $n ∈ markedNodes$ then
39:     prunePath()
40:   else
41:     $T_{callee}.add(n)$
42:     $insnToExe ← getNextInstruction()$
43:     return $T_{callee}$

Summary Nodes. Only a subset of the paths in method $M_{callee}$’s memoization tree, i.e., the feasible paths in the particular calling context from method $M_{callee}$, can be executed and should be contained in the memoization tree for method $M_{callee}$. To compactly represent these paths in method $M_{callee}$’s memoization tree, we introduce summary nodes, $S[m, p]$. These are nodes that point to a path in method $M_{callee}$, where $p$ represents a pointer to one of the leaf nodes in method $M_{callee}$’s memoization tree. Summary nodes serve as pointers to the paths in method $M_{callee}$ that are feasible in the method $M_{callee}$’s context; thus the method $M_{callee}$’s memoization tree does not need to duplicate the paths of repeated normal nodes from existing trees. Procedure $compressNode$ (Line 31) compacts a sequence of normal nodes into a summary node, which can be reverted to the original sequences of normal nodes when the memoization tree is reused for analyzing other methods.

For instance, in Figure 3 (b) nodes $n4$ and $n5$ are summary nodes, indicating $p$’s feasible paths in its calling context.
They point to the path represented by node n4 in method p’s memoization tree.

Path Conditions. Each leaf node in the tree has an associated set of path conditions characterizing the inputs that follow the path from the root to the leaf. Note that we do not have a one-to-one correspondence between paths in the memoization tree and the symbolic execution tree, the reason being that the memoization tree is more compact and can represent multiple symbolic executions.

Each tree leaf has one or more pairs of numeric path condition (PC) and heap path condition (HeapPC). Numeric path conditions depict the constraints over numeric inputs for choosing one path (see Figure 3) while heap path conditions encode heap constraints introduced when analyzing methods with parameters of reference types. We discuss the heap constraints in more detail below.

Heap Path Conditions. A Heap Path Condition is a conjunction of constraints over the heap allocated objects in the input data structures. They are generated by lazy initialization [15] during the symbolic execution of a heap manipulating method. More precisely, these constraints are generated during the symbolic execution of instructions that perform a first access to an un-initialized field (i.e. bytecodes aload, getfield and getstatic). The constraints can have the following form:

- Ref = null. Reference Ref points to null.
- Ref1 = Ref2. Reference Ref1 points to the same object in heap as reference Ref2, i.e., Ref1 and Ref2 are aliased.
- Ref ≠ null. Object reference Ref points to a symbolic object that is neither null nor any existing objects in heap, with all its fields initialized as symbolic values.

These constraints are sufficient to express all the possible aliasing scenarios in the input data structure [15]. Note that since our memoization tree only encodes the conditional branches in a method, different heap constraints can drive the program along the same path. For example, if the condition in the code checks for the input to be not null, paths characterized by non-null constraints (whether they are aliased or not) will pass that check. Thus each path in a summary tree can have one or more pairs of PC and HeapPC. We show later in this section an example of compositional analysis for heap manipulating programs.

B. Composition of Memoization Trees

Our approach uses existing memoization trees to efficiently replay the symbolic execution of the corresponding methods with respect to their calling contexts. In particular, the memoization tree of the callee method is utilized to guide part of the symbolic execution of the caller method.

Our approach first performs regular symbolic execution of method $M_{callee}$ and creates normal nodes for conditional instructions executed in method $M_{callee}$ (Lines 8–10). When the execution encounters an invocation of method $M_{callee}$, we suspend regular symbolic execution (controlled by variable toCompose), and check which paths of $M_{callee}$ are feasible in the calling context (Line 21). If a path’s PC-HeapPC pair is consistent with the current path conditions in the calling site, this path is considered feasible; otherwise, the path is infeasible and it is marked in $M_{callee}$’s memoization tree as not to be executed (Lines 24–25). Only feasible paths are considered during symbolic execution. Furthermore, constraint solving (which is typically the most expensive part of symbolic execution) is turned off during the guided execution of method $M_{callee}$ (Line 27), and is resumed when the execution returns from method $M_{callee}$ back to method $M_{caller}$ (Line 7).

Search Bound. The symbolic execution bound is fixed for a summary tree once it is built. Summary trees can be iteratively deepened (as described in our previous work [25]). To reuse a tree with a search bound, if the caller method’s bound is set to be greater than the callee method’s summary bound, our approach will turn on the constraint solver and continue as traditional symbolic execution to extend the tree for the caller method. Therefore, the set of paths explored in our compositional analysis is the same as in traditional (non-compositional) symbolic execution.

Checking Path Condition Consistency. The composition of method summaries involves checking the consistency of path conditions, to determine whether paths in the memoization tree of a called method are feasible in the current calling context. In particular, we check the consistency of the path condition in the tree with the path condition at the calling site.

Algorithm 3 checks path condition consistency. The input Mapping records the mapping between parameters of the summarized callee methods and actual inputs of the method in call site. With this mapping we convert the path conditions in the memoization trees to path conditions that refer to variables in the calling context by replacing the formal parameters of $M_{callee}$ with the actual arguments from $M_{caller}$.

Lines 1–7 check the consistency of numeric PCs. Each constraint in a path condition in the memoization tree is checked against the path condition from the calling context (we also perform some simplifications that we omit here for clarity). The conjunction of the summary PC and the context PC is checked for satisfiability using constraint solving.

Checking Heap Path Conditions. Lines 8–18 check the consistency of heap path conditions (HeapPC). Note that the heap in the calling context may be either concrete or symbolic. If it is concrete then the HeapPCs can be checked directly. However, if the current heap is symbolic, we perform an approximate consistency check for HeapPCs. The objects in method $M_{callee}$’s heap path conditions can be mapped to a concrete object in method $M_{callee}$’s calling context, or null, or a symbolic object whose fields are all symbolic values.

For each constraint in the summary heap path condition, both sides of the constraint map to lhr (left side heap reference) and rhr (right side heap reference), respectively. If both of them are null, this constraint conforms with the context heap path condition; or if both of lhr and rhr are concrete objects and they reference the same object in the calling context heap, the constraint apparently also conforms to the context heap path condition. If they are both symbolic objects, we consider it consistent as well since they are both uninitialized and would be explored by lazy initialization (Line 15). Thus, we take a
conservative approach to checking heap consistency, meaning that for the constraints that we can not decide in the current context we assume they are feasible, and we leave the lazy initialization of \( M_{\text{callee}} \) to resolve it during replay. If the conservative approach is needed we turn on the constraint solver for checking the numeric path constraints.

C. Example involving heap constraints

Consider the Swap Node example in Figure 4, where class \( \text{Node} \) implements a singly-linked list. A node has two fields \( \text{elem} \) and \( \text{next} \), representing an integer element and a reference to the next node in the list. Method \( \text{swapNode} \) destructively updates a node’s \( \text{next} \) field. Method \( \text{callSwapNode} \) creates a new concrete node \( n1 \), sets \( n1 \)'s next as the input parameter, and invokes method \( \text{swapNode} \) on \( n1 \).

We use lazy initialization [15] to analyze method \( \text{swapNode} \) and generate a memoization tree shown in Figure 5. Lazy initialization checks seven method executions that represent an isomorphism partition of the input space. However, based on conditional statements in the code, the method contains only three paths as shown in Figure 5, i.e., \( n1 \to n2 \), \( n1 \to n3 \to n4 \), and \( n1 \to n3 \to n5 \). Therefore we encode these seven different input data structures as seven pairs of numerical path condition and heap path condition, which spread across the three paths. For example, node \( n2 \) has a numeric path condition (PC) \( \text{True} \) and a heap path condition (HeapPC) \( \text{this.next==null} \). PC \( \text{True} \) indicates that no constraint on the input data structure’s integer variables is associated with this path. HeapPC implies that if input \( \text{this} \)'s field \( \text{next} \) points to \( \text{null} \), this path will be executed.

When method \( \text{swapNode} \)'s memoization tree is reused during compositional symbolic execution the paths in method \( \text{swapNode} \)'s memoization tree are checked for feasibility in this particular calling context. Figure 6 illustrates the process of checking consistency of path conditions. First, in statement 16, lazy initialization non-deterministically initializes \( n1 \).next to \( \text{null} \), or \( n1 \), or a new node with all its fields uninitialized. Then the actual parameters from the calling site are mapped to the formal parameters in the memoization tree. In this example, concrete object \( n1 \) is mapped to \( \text{this} \) in summary. For each pair of PC and HeapPC in summary tree, we check if it is satisfiable with its current input data structure from the calling site. For example, in Figure 6 one of the calling context is that \( \text{this} \) references a concrete \( \text{Node} \) object with its \( \text{elem}=0 \) and its \( \text{next} \) field pointing to another \( \text{Node} \) object with all uninitialized fields. We select two pairs of PC and HeapPC from two paths in \( \text{swapNode} \) to show how to check consistency in presence of heap operations. One is \{PC: True, HeapPC: this.next==null\} and the other one is \{PC: this.elem<this.next.elem, HeapPC: this.next!=this\}. The first one is not consistent with the calling context since \( \text{this.next} \) is \( \text{null} \); while the second one is consistent because \( \text{this.next!=this} \) conforms to the input data structure and \( \text{this.next} \) is symbolic (uninitialized) so it can be greater than or equal to \( \text{this.elem} \)'s whose concrete value is 0.

If all pairs of \( \text{PC} \) and \( \text{HeapPC} \) associated with a path in a memoization tree are checked to be inconsistent with respect to its calling context, we consider that path infeasible in the calling context. Again, the infeasible paths are marked to be pruned in symbolic execution during composition. In this example, all three paths of \( \text{swapNode} \) are feasible when invoked by method \( \text{callSwapNode} \), since every path has one or more pairs of \( \text{PC} \) and \( \text{HeapPC} \) that are consistent with calling context.

D. Discussion

Correctness. Consider a method \( M_{\text{callee}} \) that does not invoke any other methods. Then normal symbolic execution explores the same behaviors as the replay of symbolic execution of the summary tree for \( M_{\text{callee}} \). This follows from the way we construct the memoization tree, as a succinct representation of the symbolic execution tree for that method. Note also that the path conditions from the tree leaves characterize (by construction) the inputs that follow those paths.

Consider now method \( M_{\text{callee}} \) that invokes method \( M_{\text{callee}} \), for which we computed the summary tree. Whenever \( M_{\text{callee}} \) is invoked inside \( M_{\text{callee}} \) we check to see which path in \( M_{\text{callee}} \)'s summary tree can be used in the current context,
to identify the values of $M_{\text{callee}}$’s inputs, to be mapped to the corresponding symbolic values.

In practice some of the global variables should be kept concrete, as they may represent constants, instances of library classes etc. that do not need to be analyzed symbolically. This would also make the symbolic state space more manageable. Automatically identifying which fields to keep concrete is left for future work. In our current work, we assume that the developer goes into the code and annotates which fields to be kept concrete.

### E. Target Oriented Compositional Symbolic Execution

We have developed two heuristics that are specifically targeted towards covering a designated state in the program (this may be an error or a statement deep inside the program that we aim to discover quickly).

**Heuristic 1.** To quickly steer symbolic execution towards a specific target, our approach explicitly marks memoization tree nodes along paths that lead to the target, e.g., an assertion violation. In the first heuristic, our approach uses short-circuiting: if a path condition of a path that leads to an error in a summary is satisfiable in its calling context, the path does not need to be re-executed, and the error can immediately be reported and also recorded in the calling method’s summary.

Moreover, the error markings enable a directed search for errors where the replay of a memoized tree is prioritized to paths that may lead to errors: if a memoization tree checked from a top-level method has a path that terminates in an error state, the corresponding path condition can be checked for feasibility in the calling context before the other path conditions; thus, if the feasibility check succeeds, the search can report an error, thereby pruning the other memoized paths.

**Heuristic 2.** The second heuristic performs aggressive pruning of the memoized trees, by keeping only the paths that lead to the error (and discarding the rest). While incomplete, this approach turns out to be quite effective in practice, as demonstrated by the experiments in the next section.

### V. IMPLEMENTATION AND EVALUATION

**Implementation.** We implemented our compositional symbolic execution in Symbolic PathFinder (SPF), as two listeners. One builds the summary tree and the other one uses the summaries to guide symbolic execution. Whenever a conditional statement is executed (whose condition is either concrete or symbolic) we introduce a special type of choice called $\text{BranchChoice}$ (of size 1) in the SPF execution. This is a mechanism that allows us to precisely encode the conditional
### Evaluation for compositional symbolic execution

We evaluated our approach on the following Java artifacts: BankAccount, Rational, RationalRecursion, WBS, ASW, TCAS, Apollo, SwapNode, HeapOp, and LinkedList. All of these artifacts were used before for evaluating symbolic execution [2], [14], [18], [20], [21], [24], [25]. These subjects contain rich programming constructs, such as complex non-linear constraints, recursion, heap-manipulating methods, etc. that are difficult to handle with symbolic execution. The largest of these artifacts is Apollo with 2.6 KLOC in 54 classes [21]. The program contains complex non-linear floating-point constraints and it is quite challenging to analyze. To symbolically execute a configuration with two iterations, traditional SPF takes more than 4 hours to finish. We are interested in evaluating our compositional approach on such complex examples to see whether it could improve the analysis time significantly.

We have conducted experiments on the subject programs using traditional non-compositional symbolic execution (SPF) and our compositional approach (denoted here as CompoSE).

For all subjects except RationalRecursion and LinkedList, no pre-specified search depth was needed. The search depth for RationalRecursion and LinkedList is listed in Table I.

Table I shows the results of our experiments using SPF and CompoSE. We report the number of constraint solver calls, the execution time, the maximum memory (Mem) used, the number of invocations of sub-methods that have already been analyzed, and the number of nodes in compressed memoized tree using summary nodes (“CTree”) versus the number of nodes in non-compressed tree (“Tree”).

For CompoSE we show results for two steps: “Init” builds memoized summary trees for callee methods and “Comp” reuses the summaries that are already built to composition-
ally execute the target method. “Total” is the entire cost of CompoSE to analyze a method.

The methods without “Init” in Table I are the methods that we analyzed first. For these methods the number of constraint solver calls is the same for both SPF and CompoSE. This is because building a tree without any existing tree to re-use has similar cost to traditional symbolic execution, besides the cost for building and maintaining the extra tree data structure. We can see that for methods update in WBS, Main4 in Apollo, and add in LinkedList, building the summary took slightly longer than traditional SPF due to this extra work.

We can see however that for the methods that re-use the trees (marked as “Comp”) the analysis incurs fewer constraint solver calls and significantly less time than traditional SPF on most of the methods. In some cases the saving can be very significant (highlighted in bold in Table I), from 21% (simplify in RationalRecursion) to 88% (MainSymbolic in Apollo) fewer solver calls and 1.1x (main1 in WBS) to 2.21x (launch in WBS) faster. For Apollo, SPF took about 4 hours and 8 minutes to finish while CompoSE took less than 30 minutes. This result indicates that the benefits of CompoSE become more pronounced for larger programs.

CompoSE took slightly more number of solver calls than SPF in a few methods when the number of solver calls saved in composition was less than the number of solver calls in building summaries, such as method gcd in Rational, gcdRec in RationalRecursion, and a few methods in TCAS.

From the results on RationalRecursion and LinkedList it follows that as the state space to be analyzed becomes larger (i.e., by increasing the analysis depth) the savings achieved by CompoSE become more pronounced. Also naturally, the more times sub-methods are invoked the more repeated work is required so compositional approach gain more benefits. This can be verified by comparison with the numbers of sub-method invocations and the savings in Table I.

Furthermore, Table I shows that for most methods execution, CompoSE uses either the same or less memory than traditional SPF. We note however that the maximum memory reported by SPF may vary a lot due to the underlying garbage collection, and thus this comparison is less illustrative.

**Evaluation for target oriented heuristics.** We use six subjects in Table II to evaluate the effectiveness of our proposed approach with respect to the two heuristics that are target-oriented. For each subject, we selected a subset of methods to seed an assertion error in different locations. These locations are at the end of the first path, last path, and a random path (that is neither first nor last) in the method. For each run there is only one assertion error in one method. This experiment mimics scenarios where we are interested in a target located at different places in the program state space, with various degrees of difficulty in locating it.

Table II shows the comparison for SPF and the two heuristics (H1, H2). The reported numbers include the cost for building summaries of callee methods. The cases in which H1 and H2 outperform SPF are highlighted. In most cases, if the error is not in the first path of a method, our proposed approaches are better than SPF, from 1.1x to 13.4x faster. Note that for BankAccount, H2 is much faster than SPF (100x). This is due to the special structure of the program. For few other cases where methods are small (e.g., first two methods in TCAS), the overhead of building method summaries slows down our heuristics.

Conversely, if an assertion error is in the first path of a method, SPF took less time than H1 and H2. This is expected, since SPF stops immediately after it explores the first path of a method while our proposed heuristics have the overhead of building method summary trees for various methods in the call graph. Furthermore we observe that H2 always took less or same time to find the error than H1. This is because H2 is more aggressive in that it ignores more feasible error-free paths than H1.

**Threats to Validity.** The primary threats to external validity in this study involve the use of SPF for our prototype implementation, the selection of artifacts, and the use of seeded assertion errors as targets. We attempted to mitigate these threats by analyzing multiple artifacts, most of which have been used in previous studies of symbolic execution based techniques, and by seeding assertion errors at various locations in the program. These threats can be further addressed by additional evaluation using a broader range of programs and targets. The primary threat to internal validity is possible faults in the implementation of our algorithms and in SPF. We controlled for this threat by testing the implementations of the algorithms and SPF on examples that could be manually verified. Where threats to construct validity are concerned, the metrics we
selected to evaluate our approach are commonly used to measure the cost of symbolic execution based techniques.

**Discussion.** Our preliminary experiments indicate that CompoSE has little or no benefit for small programs, but the savings of CompoSE become more pronounced when the analyzed state space is larger, and for large programs (such as Apollo) the savings achieved can be very significant. Furthermore, the two target-oriented heuristics are mostly effective when the target is not easy to find (i.e., it is not on the first path). More experimentation is planned to further validate these conclusions in practice.

VI. RELATED WORK

Compositional symbolic execution has been addressed in [2], [3], [10], [12]. The work in [10] is the first to propose compositional techniques to improve a particular form of “dynamic” symbolic execution, i.e., symbolic execution performed along concrete paths [11]. This was extended in [3] with a (demand-driven) top-down approach that uses execution trees similar to ours but it is not based on replay. Instead each method summary is represented as a first-order logic formula with uninterpreted functions and the composition is performed entirely using SMT solving. A further extension [12] employs both “may” and “must” summaries, expressed with logical formulae. None of these works address composition in the presence of heap operations, for object-oriented programming. While in principle logical summaries could be computed for heap manipulating methods, the difficulty comes when reusing the summaries: one would need to reconstruct the heap according to the post condition, to continue execution.

Composition in the presence of heap operations is addressed in [2], where summaries include both logical formulae and an explicit representation of input and output heaps. The effects of the computation are stored so there is no need to “replay” symbolic execution. However, such advantage comes at a high price: the summaries are large and the composition operation is complicated, as it not only checks compatibility at the invocation point, but also it synthesizes the new state, with the new heap, to continue with after the composition of each summary. This work is based on constraint logic programming (CLP); similar synthesis of the new heap would be very expensive in a general purpose tool such as SPF. We instead construct the new heap by re-executing the code.

In a previous workshop paper [20] we described a preliminary investigation of compositional symbolic execution for Java bytecodes. We used partial evaluation, a well-established technique that aims at automatically specializing a program with respect to some of its input, to build method summaries consisting of several “path-specialized” versions of the method code. The obtained savings were not very impressive due to the expensive storing of multiple versions of the code. We use a lighter-weight approach here, since the method summaries only encode the choices taken along each path. This is sufficient for method replay.

Compositional interprocedural analysis has been extensively studied [13] with recent techniques focusing on handling heap operations [6], [9]. The compositional shape analysis from [6] uses separation logic and assigns a collection of Hoare triples to each procedure that is analyzed separately, in bottom-up fashion. The triples provide an over-approximation of data structure usage. We provide an under-approximation, as is typical in symbolic execution, and we do not use a logical encoding for our summaries. The work in [9] presents a bottom-up, summary-based heap analysis that uses abstract points-to graphs extended with constraints to encode heap summaries. The work does not address computing summaries about shapes of data structures as in [6] but guarantees a higher level of precision.

A recent related target-oriented technique is presented in [7], although it is done in the context of bounded model checking not symbolic execution, and it performs weakest preconditions calculations instead of forward computations. Furthermore it does not handle heap operations but it is targeted towards the properties to check. Other target-oriented symbolic execution techniques are presented in, e.g., [5], [17], [23]. A thorough empirical comparison with these techniques is future work.

In previous work we developed memoized symbolic execution (Memoise) [25] for the efficient re-application of symbolic execution in different scenarios such as iterative deepening and regression analysis. Similar to the approach here, Memoise stores (on disk) the key elements of symbolic execution in a tree data structure, and uses that tree during re-execution. However the approach is not compositional. The technique that we present in this paper naturally extends this previous work, enabling memoization and retrieval of trees at a finer granularity, resulting in more efficient analysis.

VII. CONCLUSION

We presented a new approach for compositional symbolic execution. The approach summarizes each analyzed method as a memoization tree that captures the crucial elements of symbolic execution, and leverages these trees to efficiently replay the symbolic execution of the corresponding methods in different calling contexts. Our approach offers a natural way to compose in the presence of heap operations, which cannot be dealt with by previous work that uses logical formulas as summaries for compositional symbolic execution. We also presented two heuristics for the efficient treatment of error traces. Preliminary experimental evaluation based on implementation in Symbolic PathFinder shows promising results. We believe compositional analysis holds a key to scalable symbolic execution. In future work, we plan to evaluate our approach on larger programs as well as to further optimize our algorithms. We also plan to incorporate loop invariants for the succinct summarization of looping methods.

**Acknowledgments**

This work was funded in part by the National Science Foundation (NSF Grant Nos. CCF-1319858, CCF-1319688, CCF-0845628, and CCF-1329278).
REFERENCES