The Thesis Committee for Rui Qiu
certifies that this is the approved version of the following thesis:

Compositional Symbolic Execution with Memoized Replay

APPROVED BY

SUPERVISING COMMITTEE:

Sarfraz Khurshid, Supervisor

Guowei Yang, Co-Supervisor

Compositional Symbolic Execution with Memoized Replay

by

Rui Qiu, B.E.

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2014

Acknowledgments

I would like to express my deepest appreciation to my supervisor Professor Sarfraz Khurshid and my co-supervisor Professor Guowei Yang, for their genius guidance and generous help. I could not have imagined having better supervisors to guide me through my master study.

Besides my thesis committee members, I would like to specifically thank Corina Păsăreanu for her mentorship and supervision throughout this project. Corina conceived the idea of using memoization trees for compositional symbolic execution and her deep insights into symbolic execution were essential to and immensely helpful in this project's success.

I would also like to thank my lab-mates Lingming Zhang and Shiyu Dong for their support and encouragement during the past year. A special thanks to my family. Words cannot express how grateful I am for everyone's support and help.

This thesis is in part based on a paper entitled "Compositional Symbolic Execution with Memoized Replay" co-authored by Rui Qiu, Guowei Yang, Corina S. Păsăreanu, and Sarfraz Khurshid, currently under submission for peer-review.

This work was funded in part by the National Science Foundation (NSF Grant Nos. CCF-0845628, CCF-1319688, and CNS-0958231).

Compositional Symbolic Execution with Memoized Replay

Rui Qiu, M.S.E
The University of Texas at Austin, 2014

Supervisor: Sarfraz Khurshid
Co-Supervisor: Guowei Yang

Symbolic execution is a powerful, systematic analysis 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 thesis 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 an efficient treatment of error traces by short-circuiting the execution of paths that lead to them. Our preliminary experimental evaluation based on a prototype implementation in Symbolic PathFinder shows promising results.

Table of Contents

List of Tables

List of Figures

Chapter 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 or security analysis, continuous testing and program repair [5, 17, 23]. 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 in the past to scale up static analysis and software verification techniques [6, 7, 9, 12], including symbolic execution [4, 10, 11]. 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 behaviour of the unit. Whole-program analysis results are then obtained by

incrementally composing and utilizing the previously built summaries.

This thesis 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 trees succintly summarize the *feasible* paths through the method and it does not explicitely 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 [4, 10, 11]. When composing a method summary with the actual calling context, we first perform a partial check of the heap constraints on the (pos-

sibly 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.

For error detection, our approach explicitly marks memoization tree nodes that correspond to *error* states, e.g., due to an assertion violaion. As a result, our approach enables an efficient treatment of error traces by short-circuiting: if a path condition for a path that leads to an error state in a summary is feasible in the current calling context, the whole memoized 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 before the other path conditions; thus, if the feasibility check succeeds, the search can report an error, thereby pruning the other memoized paths.

Besides the obvious benefits of increased scalability, our work also 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 bottom-up in parallel and

memoized path conditions can be checked for the current calling context in parallel.

We make the following contributions:

- **Memoization trees.** We introduce the idea of performing compositional symbolic execution using memoization trees;
- **Summarize and compose.** We define the core algorithms that embody compositional symbolic execution using memoization trees;
- **Program heap.** Our approach supports compositional symbolic execution in the presence of operations on the program heap;
- **Experiments.** We implemented our approach into a prototype tool that builds on top of the Symbolic PathFinder tool [18]. We describe initial experimental results, which show the promise our approach holds.

The remainder of this thesis is organized as follows: We introduce background of symbolic execution in Chapter 2. Chapter 3 illustrates examples for both traditional symbolic execution and our proposed memoized tree based approach. In Chapter 4 we present our proposed approach in detail and in Chapter 5 we show evaluation results with our prototype implementation. Related work is discussed in Chapter 6. We conclude in Chapter 7.

Chapter 2

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 program in Figure 2.1 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 start point of symbolic execution. Figure 2.2 shows the complete symbolic execution tree of method q. Initially, PC is true

```
1  int p(int x, int y) {
2    if(x > y) {
3      x--;
4    } else {
5      y++;
6    }
7    if(x == y) {
8      return x;
9    } else {
10     return y;
11   }
12 }
13
14 int q(int a, int b) {
15   if(a > b) {
16     return p(a + 1, b - 10);
17   } else {
18     return p(b + 1, a - 10);
19   }
20 }
```

Figure 2.1: Caller method q and callee method p

and a, 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 16 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 all possible paths are explored. Whenever PC is updated an off-the-shelf constraint solving decision is called to check if the updated PC is satisfiable. If it is not satisfiable, the execution backtracks to previous PC

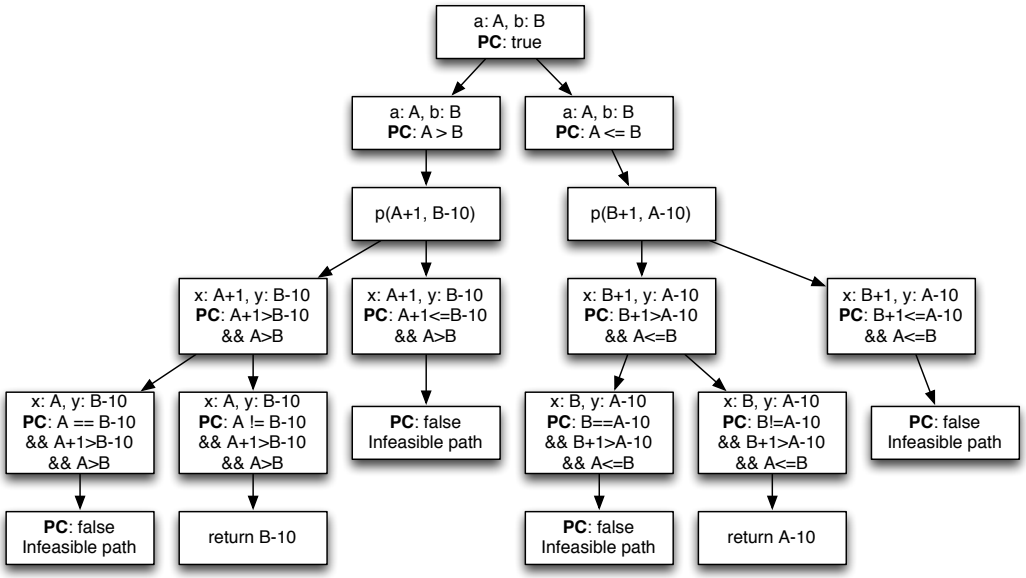

Figure 2.2: Symbolic execution tree for method q

and continues execution. 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) is an open source tool that performs symbolic execution for Java programs at the bytecode level. SPF uses lazy initialization [15] to handle dynamic input data structures (e.g., lists and trees). The components of the programs 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 `f` of type `T` is accessed in `m` for the first time, SPF non-deterministically sets `f` 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.

Chapter 3

Example

This section uses two example programs to informally illustrate our compositional symbolic execution approach.

When traditional symbolic execution is applied on the method q shown in Figure 2.1, method p in the same Figure is symbolically executed twice, since both branches of the conditional statement at line 15 are feasible. The cost of method p's "re-execution" can be reduced by compositional symbolic exeuction, where we first build a memoization tree of method p, and then efficiently perform symbolic exeuction of method q by replaying the symbolic execution of p in the two calling contexts using p's memoization tree.

Figure 3.1 shows the memoization tree for method p. 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 $n1$ indicates that the true (1) branch of the conditonal 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

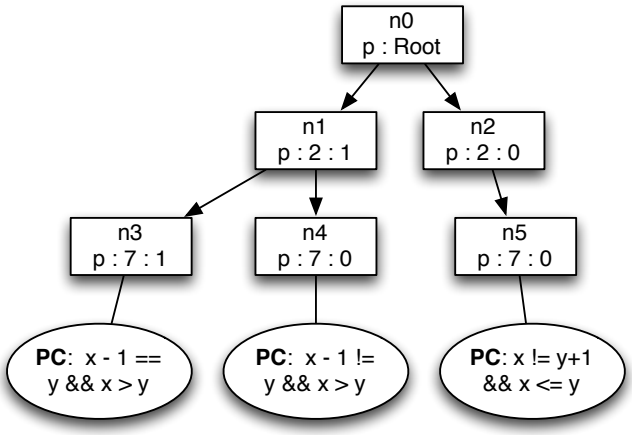

Figure 3.1: Tree built for method p

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 16, method p's actual inputs are $a + 1$ and $b - 10$, and its formal arguments are x and y. So we map $a + 1 \rightarrow x$ and $b - 10 \rightarrow y$. 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 3.1, for instance, the transformed path condition $(a + 1)! = (b - 10) + 1 \&\& (a + 1) <= (b - 10)$ after replacing the formal ar-

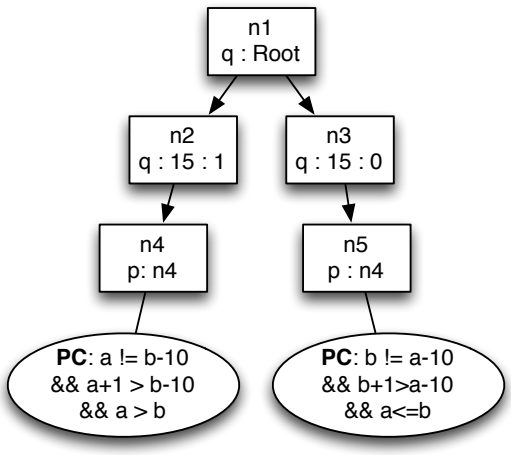

Figure 3.2: Tree built for method q

guments, 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 3.2 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 3.2 are summary nodes. In node $n4$, "$[p : n4]$" points to the path ended at node $n4$ ($n0 \rightarrow n1 \rightarrow n4$) in method p's memoization tree.

As shown in Figure 2.2, in traditional symbolic execution, whenever the path condition is updated it is checked for satisfiability using the underlying

```
1  class Node {
2    int elem;
3    Node next;
4
5    Node swapNode() {
6      if(next != null) {
7        if(elem > next.elem) {
8          Node t = next;
9          next = t.next;
10         t.next = this;
11         return t;
12       }
13     }
14     return null;
15   }
16
17   static Node callSwapNode(Node n) {
18     Node n1 = new Node();
19     n1.next = n;
20     return n1.swapNode();
21   }
22 }
```

Figure 3.3: Swap Node example

constraint solver. So traditional symbolic execution makes 10 constraint solver calls in total for symbolically executing method q. While our approach takes 8 constant 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 16 and line 18).

Consider the Swap Node example [15] in Figure 3, where a class *Node* is declared to implement a singly-linked list. The *Node* has two fields *elem* and *next*, representing its integer element and a reference to its next node respectively. Method *swapNode* destructively updates a node's *next* field. Method *callSwapNode* creates a new concrete node $n1$, sets $n1$'s next as the input parameter, and invokes method *swapNode* on $n1$.

We use *lazy initialization* [15] to analyze method *swapNode* and gen-

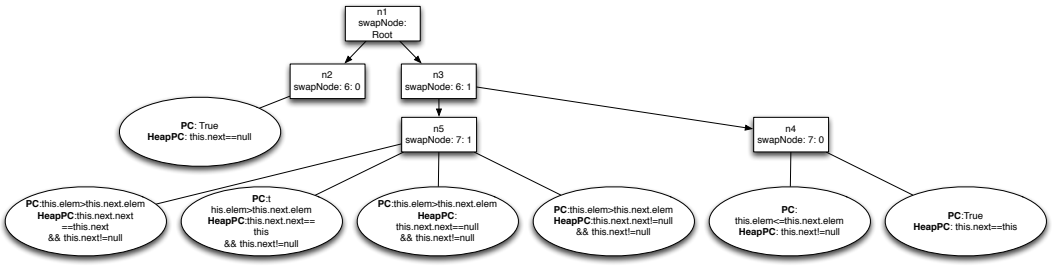

Figure 3.4: Tree built for method *swapNode*

erate a memoization tree shown in Figure 3.4. 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 3.4, i.e., $n1 \rightarrow n2$, $n1 \rightarrow n3 \rightarrow n4$, and $n1 \rightarrow n3 \rightarrow n5$. 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 (NumPC) $True$ and a heap path condition (HeapPC) $this.next == null$. NumPC $True$ indicates that no constraint on the input data structure's integer variables is associated with this path. HeapPC implies that if input *this*'s field *next* points to *null*, this path will be executed.

When method *swapNode*'s memoization tree is reused during compositional symbolic execution, like what we did before, the paths in method *swapNode*'s memoization tree are checked for feasibility in the particular calling context. Figure 3.5 illustrates the process of checking consistentency of path conditions. First, in statement 19, lazy initialization nondeterministically initializes $n.next$ to null, or $n1$, or a new node with all its fields unini-

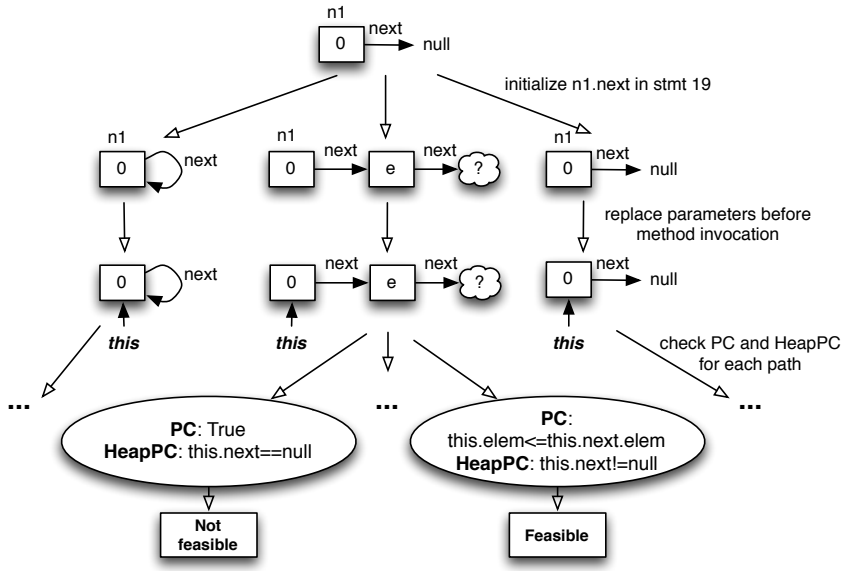

Figure 3.5: The process for checking path conditions when reusing summary tree

tialized. Then the actual parameters from the calling site are mapped to the formal parameters in the moization tree. In this example, concrete object $n1$ is mapped to *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 3.5 one of the calling context is that *this* references a concrete $Node$ object with its $elem = 0$ and its $next$ field pointing to another $Node$ object with all uninitialized fields. We select two pairs of PC and HeapPC from two paths in $swapNode$ to show how to check consitency 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 $this.next$ is not $null$; while the second one is consistent because $this.next != this$ conforms to the input data structure and $this.next.elem$ is symbolic (uninitialized) so it can be greater than or equal to $this.elem$ whose concrete value is 0.

If all pairs of PC and 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 $swapNode$ are feasible when invoked by method $callSwapNode$, since every path has one or more pairs of PC and HeapPC that are consistent with the calling context.

Chapter 4

Compositional Symbolic Execution with Memoized Replay

4.1 Overview

We propose a context-insensitive compositional approach where 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*.

When deciding on the order to analyze the methods in a given system, two main strategies can be followed. A context-sensitive or top-down approach may be adequate if one wants to compute only the strictly necessary information. However, this approach does not guarantee that reusability of summaries

is always possible, and summaries may need to be recomputed. On the other hand, a context-insensitive or bottom-up approach ensures that the computed summaries can always be reused, at the price of computing summaries larger than necessary in some cases. We follow the latter strategy in our framework.

Let us assume that program P consists of an entry method M_0 and a set of methods $M_1, M_2, ..., M_k$ that belong to M_0's method invocation hierarchy. In the method invocation hierarchy, each leaf method, i.e., the method that has no invocation of other methods, is analyzed first, and its feasible paths and corresponding path constraints are summarized in a memoization tree. Then each method that directly invokes these leaf methods is analyzed by leveraging the already built memoization trees of the invoked methods (leaf methods). Incrementally, we symbolically analyze methods from bottom up in the method invocation hierarchy, and build their memoization trees leveraging the existing memoization trees, until the entry method M_0 is analyzed.

Our approach works on a method invocation hierarchy that has no cycles in the coresponding call graph. If a method node in the call graph has an edge to itself, i.e., the corresponding method is a recursive method, we simply ignore this recursive method invocation edge during processing, and we analyze the method up to the given bound. For other method invocation chain that contains a loop, we break the loop by ignoring one random edge in that loop to form an acyclic call graph. This simple approach is sufficient for error detection or test case generation.

Algorithm 1 describes the pseudocode for our overall approach. Proce-

dure `getStaticCallGraph`(P, M_0) (Line 2) creates a static call graph of program P rooted from method M_0, and procedure `convertToAcyclic` (Line 3) removes cycles in the static call graph. Procedure `getLeafNodes` (Line 4) identifies the methods corresponding to leaf nodes in the invocation hierarchy. Furthermore, our approach checks whether the analyzed method has conditional statements using the helper method `containsConditionalStatement` (Line 7), and skips the analysis of the method that does not contain any conditional statements, since the method without conditional statement contains only one path and reusing the memoization tree of such method would not gain much benefits during compositional symbolic execution. Procedure `analyzeMethod` (Line 8) performs a bounded symbolic execution for a given method and builds its memoization tree using existing memoization trees of its callee methods. Procedure `getNodesToProcess` (Line 12) returns the nodes in the call graph whose correponding methods have not been processed yet the methods that they invoke have already been processed.

In this way, our approach symbolically executes and builds method summary trees for each method in S_M according to their orders in call graph. And at last analyzes entry method M for program P. In contrast with traditional symbolic execution, our approach incrementally analyzes each methods from leaf nodes of call graph up to the entry method of the program rather than symbolically executing whole program from method M.

Algorithm 2 shows the pseudo-code for *analyzeMethod* for building and composing method summary trees. We describe some of its key elements

```
Input: Program P, Entry method M_0
Output: A set of method summary trees S_T for methods in P
1  S_T ← ∅
2  CallGraph cg ← getStaticCallGraph(P, M_0)
3  cg.convertToAcyclic()
4  Set<MethodNode> s ← cg.getLeafNodes()
5  while s ≠ ∅ do
6      foreach Method m ∈ s do
7          if m.containsConditionalStatement() then
8              SummaryTree t ← analyzeMethod(m, S_T)
9              S_T.add(t)
10         end
11     end
12     s ← s.getNodesToProcess()
13 end
14 return S_T
```

Algorithm 1: Overview algorithm for compositional symbolic execution with memoized replay

in more detail below.

In the following, we assume two methods M_{caller} and M_{callee}, the running example in this section, where method M_{caller} invokes method M_{callee}, and the memoization tree T_{callee} for method M_{callee} is already built in a previous run.

4.2 Memoization Tree Construction

A memoization tree is a recursive tree data structure that captures the crucial elements of symbolic execution.

Each leaf node in the tree has an associated set of path conditions characterizing the inputs that follow the path from s_0 to the leaf.

In the memoization tree, there are two types of nodes: *normal nodes* and *summary nodes*. At a high level, for each method M_{caller} invoking method M_{callee}, 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 invokation point from M_{caller}.

A normal node, **N[m : offset : choice]**, is a node that represents a choice taken at a conditional statement and it encodes the following: m is the name of the method that conditional statement belongs to, *offset* is the instruction offset of the conditional statement, and *choice* is the choice taken by the execution (we use integer "1" for true branch and integer "0" for false branch). A normal node is created whenever a branch of a conditional statement is executed. For example, in Figure 3.1, node $n2$ has a tuple $[p : 2 : 0]$, indicating that in method p instruction with offset 2 ("$if(x > y)$") 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 memoized tree by traversing the tree from the root to a leaf. We thus can use the leaf nodes to represent their correponding paths. For example, in Figure 3.1, 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 in the summary 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 summary generation may become concrete due to concrete inputs from the calling context; we can not distinguish that from a condition that was concrete also in the summary. We therefore chose to record *all* the conditions in the summary tree and use the tree to guide the execution of all the conditions. This is described in lines 8-10 in *analyzeMethod*.

When a method invokes another method only a subset of the paths in the callee's memoization tree, i.e., the feasible paths in the particular calling context, can be executed and should be contained in the memoization tree for the caller method. To compactly represent these paths in the caller's memoization tree, we introduce *summary nodes*, **S[m, p]**, a node that points to a path in method m (a callee method), where p represents a pointer to one of the leaf nodes in m's memoization tree. Summary nodes serve as pointers to the paths in the callee method that are feasiblle in the caller's context; thus the caller method's memoization tree does not need to duplicate the paths of repeated normal nodes from existing trees. Procedure *compressNode* (Line 37) 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.

Thus a memoization tree for a leaf method in the call graph contains only normal nodes, since such a method does no invoke other methods and can not reuse other existing memoization trees; while a memoization tree for a upper level method in the call graph can contain both summary nodes and

normal nodes.

For instance, in Figure 3.2 nodes $n4$ to $n9$ are summary nodes, indicating p's feasible paths in its calling context. Node $n4$ in Figure 3.2 points to the path represented by node $n3$ in method p's memoization tree.

4.3 Path Conditions

For each path in a memoization tree we associate the set of path conditions characterizing the symbolic execution paths that follow the same path in the tree. Note that we do not have a one-to-one correspondance between paths in the memopized tree and the symbolic execution tree, the reasoning being that the memoization tree is more compact and can represent multiple symbolic executions.

This set of path conditions consists of one or more pairs of numeric path condition (PC) and heap path condition (HPC). Numeric path conditions depict the constraints over numeric inputs for choosing one path while heap path conditions are introduced with heap allocated objects in the input data structure. HPC is generated by *lazy initialization* [15] during the symbolic execution of a heap manipulating method and it consists of a conjunction of three forms of constraints:

- $Ref = null$. Object reference Ref points to $null$.
- $Ref_1 = Ref_2$. Object reference Ref_1 points to the same object in heap as object reference Ref_2, i.e., Ref_1 and Ref_2 are aliased.

- $Ref \neq null$. 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.

Note that since our memoization tree only encodes the conditional branches in a method, different heap constructs can drive the program to follow one same path. Thus each path in a summary tree can have one or more pairs of PC and HPC

For example, in Figure 3.4, four pairs of HeapPC and Numeric PC are associated with node $n5$. These four pairs represent four symbolic execution runs that all drive the method to follow the same path. The first HeapPC associated with node $n5$ is $this.next.next{==}this.next$ && $this.next! = null$. This HeapPC implies that input $this$'s $next$ field references a symbolic $Node$ object N whose $next$ field references back to N itself. Similarly, the second HeapPC $this.next.next == this$ && $this.next! = null$ represents that input $this$ references to a $Node$ object that has its $next$ field initialized as an object N whose integer field $elem$ is symbolic (not initialized) and field $next$ references back to input $this$.

4.4 Memoization Tree Composition

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 exection of the caller method.

Our approach first performs regular symbolic execution of method M_{caller} and creates normal nodes for conditional instructions executed in method M_{caller} (lines 8-10). When the execution encounters invocation of method M_{callee}, we suspend regular symbolic execution (controlled by variable $toCompose$), and check which paths of M_{callee} are feasible in this calling context. If a path's PC-HPC 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 the reused memoization tree as not to be executed due to its infeasibility. Only feasible paths are executed during symbolic execution. Furthermore, constraint solving is turned off during this guided execution of method M_{callee}, and is resumed when the execution returns from method M_{callee} back to method M_{caller}.

While a method that has a summary tree is invoked in a caller method, a subset of callee method's paths in this particular calling context may be feasible and the rest of the paths may be infeasible. To represent a feasible path in the callee's summary tree, we introduce *summary node*, $N_s[Method, Pointer]$, a node that points to a path in $Method$, where $Pointer$ designates one of the nodes in the end of a path in an existing summary tree. *Summary node* serves as a pointer to a certain path in a callee method so caller method's summary tree does not have a path of normal nodes same from existing trees. Thus the total number of nodes in a summary tree can be reduced. For example, in Figure 3.2 nodes $n4$ to $n9$ are summary nodes, indicating feasible paths in

method p when it is invoked in method q. With the $Pointer$ value in *summary node*, we can convert summary node to a path of nodes as they are in callee method's summary tree.

We can traverse from the root node of a summary tree to a leaf node to form a path that has been executed during symbolic execution. Every such path in the tree associates a set of *numeric path conditions* (PC) and *heap path conditions* (HPC) with it. PC is a set of boolean constraints in terms of symbolic numeric inputs. HPC is a set of conjunctions of constraints over the heap allocated objects in the input data structure. These constraints on heap objects are generated by lazy initialization during symbolic execution when there is heap operations.

A node in the summary tree can be marked according to different applications needs. A subset of leaf nodes in summary tree can be marked as *boundary nodes* as they are the nodes where execution backtracks when search depth limit is reached. These *boundary nodes* can be used when iterative deepening is performed. A subset of nodes can also be marked as infeasible choices when the method is invoked in a certain calling context. Or whenever an error or an exception is thrown, the corresponding path end node can be marked as an *error node* to indicate that an error occurs in this certain path. We can short-circuit the paths with an *error node* to report quickly that an error will occur under a calling context if that path is feasible in this context.

4.5 Checking Path Condition Consistency

The compostion of method summaries involves checking the consitency 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 memoization tree with the path condition at the calling site.

During the process of building the summary tree for method T_{callee}, if a method m_k that already has a summary tree is invoked by method T_{callee}, we suspend traditional symbolic execution and retrieve the callee method m_k's summary to guide execution. Normally a subset of paths are feasible in the summary tree of method m_k so we will first check which paths are feasible in the calling context of method T_{callee}. The infeasible paths will be marked in the summary tree and we use this marked tree to guide symbolic execution, i.e., if an infeasible node's choice is chosen, execution will request backtrack. To check whether a path in a summary is feasible, we build a mapping between the actual parameters and the formal parameters in method summary. For instance, in *swapNode* example, we map concrete object $n1$ to the parameter *this* in summary of method *swapNode*. In example of method q invokes method p, $a+1$ and $b-10$ are mapped to x and y respectively in the first call site.

Algorithm 3 shows how we check path condition consistency. The input `Mapping` records the mapping between parameters of the summarized caller methods and actual inputs of the method in call site. With this mapping

we can easily convert the path conditions in the memoization trees to path conditions that refer to variables in the calling context by replacing the formal arguments of M_{callee} with the actual arguments from M_{caller}.

Lines $20-29$ check the consistency of numeric PCs. Each constraint in a path condition in the memoization tree is checked against the context path condition. If the negation of the constraint is included in the context path condition, this path cannot be feasible in the calling context and thus the procedure returns false; otherwise, we add the constraint to the context's path condition. When we have processed all the constraints in the summary PC, the conjunction of the summary PC and the context PC is checked for satisfiability using constraint solving.

Lines $5-19$ check the consistency of heap path conditions (HPC). Note that the heap in the calling context may be either concrete or symbolic. If it is concrete then the HPCs can be checked easily. However, if the current heap is symbolic, we can only perform an *approximate* consistency check for HPCs. The objects in method M_{callee}'s heap path conditions can be mapped to a concrete object in method M_{caller}'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 *lhc* and *rhc*, respectively. If both of them are *null*, this constraint conforms with the context heap path condition; or if both of *lhc* and *rhc* 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. Thus, we take a *conservative* approach to check 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_{callee} to resolve it during replay. If conservative approach is needed we will turn on the constraint solver for checking the unsure path constraints.

For example, in *swapNode* example in Figure 3.5, when the calling context input of method *swapNode* is $n1$ and $n1$'s next field points to a symbolic object, we consider the second Heap PC ($this.next! = null$) in the graph consistent because it also indicates that the *next* field references a symbolic object. Since one of the pairs of Heap PC and numeric PC that related to the path of node $n5$ is satisfiable in the calling context, this path will be marked as feasible. However, when this path is re-executed during method composition, we still need to turn on the constraint solver to check the feasibility of this path as a conservative approach is used.

4.6 Handling of Errors

Let us assume that we are checking method M_{caller} for any unhandled exceptions or assertion errors, and method M_{callee} contains an error under a particular path condition. Traditional symbolic execution starts executing method M_{caller} and continues to method M_{callee} until the path that has an error in M_{callee} is explored. We propose to speed-up this checking by short-

circuiting the paths in method M_{callee} to report the error. In particular, we introduce *error nodes*, which are the leaf nodes of paths in T_{callee} that can lead to errors. When method M_{callee} is invoked, we check whether there is a feasible path ending in an error node; if so, we immediately report the error without replaying M_{callee} for that path.

The *correctness* of our proposed approach is based on the following observation: statement s is explored by full symbolic execution *iff* s is explored by compositional symbolic execution.

Input: Method M_{caller}, Set¡Memoization Tree¿ S_T
Output: Memoization tree T_{caller} for method M_{callee}

```
1  boolean toCompose ← false
2  Memoization Tree T_caller ← new empty Memoization Tree
3  Memoization Tree t ← null
4  Instruction insnToExe ← getNextInstruction()
5  while insnToExe! = null do
6      if !toCompose then
7          turnOnConstraintSolver
8          if type(insn) == ConditionalInstruction then
9              Normal Node n ← createNode(insn)
10             T_caller.add(n)
11         end
12         if type(insnToExe) == InvokeInstruction then
13             if invokedMethod(insnToExe) ∈ S_T then
14                 toCompose ← true
15                 t ← S_T.getSummary(insnToExe)
16                 Mapping mapping ← createMapping()
17                 foreach Path path ∈ t.getPaths() do
18                     SPC spc ← path.getSummaryPC()
19                     boolean isConsistent ← false
20                     foreach PCPair pcp ∈ spc do
21                         if check(pcp, context, mapping) then
22                             isConsistent ← true
23                             break
24                         end
25                     end
26                     if isConsistent == false then
27                         markNodes(path)
28                     end
29                 end
30             end
31         end
32     else
33         turnOffConstraintSolver
34         if type(insnToExe) == ReturnInstruction then
35             if backToCaller(insnToExe) then
36                 toCompose ← false
37                 T_caller.compressNode()
38                 turnOnConstraintSolver
39             end
40         end
41         if type(insnToExe) == ConditionalInstruction then
42             Node n ← t.getNextNode()
43             if n ∈ summaryNodes then
44                 t.decompressNodes(n)
45             end
46             n ← t.getNextNode()
47             if n ∈ markedNodes then
48                 prunePath()
49             else
50                 T_caller.add(n)
51             end
52         end
53     end
54     insnToExe ← getNextInstruction()
55 end
```

Algorithm 2: Procedure *analyzeMethod* for building memoization trees

Input: PCPair pcp, Context context, Mapping mapping
Output: true for satisfiable case, or false for unsatisfiable case

```
1  NPC npc ← pcp.getNPC()
2  NPC contextnpc ← context.getNPC()
3  HPC hpc ← pcp.getHPC()
4  HPC contexthpc ← context.getHPC()
5  foreach Constraint hc ∈ hpc do
6  |  Comparator comp ← hpc.getComparator()
7  |  if comp == "! =" then
8  |  |  continue
9  |  end
10 |  Referecne lhc ← hpc.getLeftSideRef()
11 |  Reference rhc ← hpc.getRightSideRef()
12 |  if (lhc == nullRef ∧ rhc == nullRef)
13 |  |  ∨(lhc == object ∧ rhc == object)
14 |  |  ∨(lhc == null ∨ rhc == null) then
15 |  |  continue
16 |  else
17 |  |  return false
18 |  end
19 end
20 foreach Constraint c ∈ npc do
21 |  c ←replaceVariables(c, mapping)
22 |  if c ∈ contextnpc ∨ !c.isConcretized() then
23 |  |  continue
24 |  end
25 |  if ¬c ∈ contextnpc then
26 |  |  return false
27 |  end
28 |  contextnpc ← contextnpc.addConstraint(c)
29 end
30 boolean pcSatisfied ← contextnpc.solve()
31 if pcSatisfied == false then
32 |  return false
33 end
34 return true
```

Algorithm 3: PC and HeapPC consistency checking

Chapter 5

Implementation and Evaluation

In this section we describe our implementation and the experiments we have conducted to show the effectiveness and benefits of our approach.

5.1 Implementation

We implement our compositional symbolic execution with memoized replay on top of Symbolic PathFinder (SPF), an open source tool that performs symbolic execution for Java programs at the bytecode level. SPF extends the analysis engine framework of Java PathFinder (JPF) tool-set. JPF implements an extensible and customizable Java Virtual Machine (JVM) that can be used for model checking. By executing Java bytecode instructions, JPF can generate, store, and search states. JPF implements a form of state space reduction that lumps together sequential transitions which are broken only if a non-deterministic choice is encountered. *Listeners* in JPF monitor and interact with the execution of the analyzed Java program.

SPF implements a non-standard interpretation of Java bytecode. Symbolic values can be propagated through method input, variables, operands, and etc. Whenever a conditional instruction is executed symbolically, SPF uses a

path condition (PC) choice generator to encode a non-deterministic choice of with branch to take. Each choice is associated with a path constraint that can be checked for satisfiability using off-the-shelf decision procedures or constraint solvers. If the constraint is satisfiable, the search continues; otherwise, the search backtracks.

We implement our proposed approach on top of (SPF), as two JPF listeners. One builds the summary tree and the other one utilizes existing summaries to guide symbolic execution. Whenever a conditional statement is executed (whose condition is either concrete or symbolic) we introduce of a new special type of choice called " *BranchChoice*" (of size 1) in the SPF execution. This is a mechanism that allows us to precisely encode the conditional instruction's location and the choice it takes, and to "break" the transitions that JPF lumps together to introduce new points to which the tool can backtrack. When building the summary tree, the listener monitors whenever a new *BranchChoice* is created, i.e. a conditional instruction is executed, and adds a *normal node* to the tree as a child to the current node. When SPF backtracks from current *BranchChoice* to the previous one in the search, the current node backtracks to its parent node too to keep the states of the program and current node of the tree in sync.

We use the Eclipse Java Development Tool (JDT) [1] to build a static call graph for a given program. From that, we prune the call graph rooted

[1] http://www.eclipse.org/jdt/

from a specified entry method. Any methods that are not reachable from this entry method will not be analyzed since they are irrelevant to building the summary tree for the target entry method.

5.2 Artifacts

We selected ten artifacts for our experiments: `BankAccount`, `WBS`, `ASW`, `TCAS`, `Apollo`, `Rational`, `Swap Node`, `Foo example`, net data structures, and `Sorted Link List`. All of these artifacts are small Java program used in previous work on evaluating symbolic execution techniques [3, 13, 17, 19, 21, 24, 25]. The largest of these artifacts is Apollo with 2.6 KLOC in 54 classes.

5.2.1 BankAccout

The bank account example in Figure 5.1 has been used in previous work [13] to illustrate method sequence generation using symbolic execution and evolutionary testing. The example implements a bank account service, and contains the `deposit` method and the `withdraw` method. are used to deposit money in and withdraw money from the account, respectively. If the amount to be withdrawn is greater than the account balance, or if the number of withdrawals (`numberOfWithdrawals`) completed is greater than or equal to a fixed quantity (5), an error message is again printed and the method exits.

```
1  public class BankAccount {
2     private int balance;
3     private int numberOfWithdrawals;
4     public void deposit(int amount) {
5       if (amount > 0)
6         balance = balance + amount;
7     }
8     public void withdraw(int amount) {
9       if (amount > balance) {
10        printError();
11        return;
12      }
13      if (numberOfWithdrawals >= 5) {
14        assert false;
15        printError();
16        return;
17      }
18      balance = balance - amount;
19      numberOfWithdrawals++;
20   }
```

Figure 5.1: A bank account example

5.2.2 WBS

Wheel Brake System (WBS) is a synchronous reactive component from the automotive domain. This method determines how much braking pressure to apply based on the environment. The Java model is based on a Simulink model derived from the WBS case example found in ARP 4761 [14, 20]. The Simulink model was translated to C using tools developed at Rockwell Collins and manually translated to Java. It consists of one class and 231 lines of code. We have used this example before in the context of regression analysis using symbolic execution [17].

5.2.3 ASW

The Altitude Switch (ASW) program is a synchronous reactive component from the avionics domain. It turns power on to a Device of Interest (DOI) when the aircraft descends below a threshold altitude above ground level (AGL). It was developed as a Simulink model, and was automatically translated to Java using tools developed at Vanderbilt University [22].

5.2.4 TCAS

Traffic Anti-Collision Avoidance System (TCAS) is a system to avoid air collisions. Its code in C together with 41 mutants are available at SIR repository [2]. TCAS was converted to Java manually and has 143 line of code.

5.2.5 Apollo

The Apollo Lunar Autopilot is a Simulink model that was automatically translated to Java using Vanderbilt tools. The translated Java code has 2.6 KLOC in 54 classes. The Simulink model was created by an engineer working on the Apollo Lunar Module digital autopilot design team. The goal was to study how the model could have been designed in Simulink, if it had been available in 1961. The model is available from MathWorks6. It contains both Simulink blocks and Stateflow diagrams and makes use of complex Math functions (e.g. Math.sqrt). The code has been analyzed before using Symbolic PathFinder with the Coral solver [21]. These five artifacts were used

in previous work to analyze *memoized symbolic execution* [25]. We manually converted the code to Java. The Java version has 143 lines of code.

5.2.6 Rational

Rational is a case study program that stresses the use of linear constants and it was used in previous work on compositional symbolic execution [3]. It contains 4 methods: `abs, gcd, simplify` and `simp`. Method `abs` computes the absolute value of an integer and method `gcd` computes the greatest common divisor of two integers using `abs`. Method `simplify` invokes method `gcd` and method `simp` invokes method `simplify` multiple times.

5.2.7 Swap Node

We modified the example in Figure 3 to add a boolean variable as an input to method `swapNode` to form a new method called `swapNodeWithBoolean` (in Figure 5.2), which updates the input data structure according to the boolean value and the `elem` field of the `Node` object. We also created another two methods `callTwice` and `call3Times` that invoked method `swapNodeWithBoolean` 2 and 3 times respectively.

5.2.8 Foo example

Program `Foo` is a small case study from [19]. It was created to illustrate the the code slicing in the presence of heap operations. The program has a class named `Foo` and it contains two methods `q` and `m` where `m` invokes `q` several

```
1  public Node swapNodeWithBoolean(boolean b) {
2    if (next != null) {
3      if ((elem > next.elem && b) || (elem <= next.elem && !b)) {
4        Node t = next;
5        next = t.next;
6        t.next = this;
7        return t;
8      }
9    }
10   return this;
11 }
```

Figure 5.2: swapNodeWithBoolean example

times. Both methods involve operations of updating heap objects.

5.2.9 `net` data structures

The `net` data structures is a real-life example borrowed from the `net.datastructures`[2] Java package , which is an educational collection of Java interfaces and classes that implement fundamental datastructures (e.g., search trees and graphs) and algorithms (e.g. sorting and traversal). We selected method `swapElements` from class `NodePositionList` as the entry method. Method `swapElements` takes as input two `DNode` objects, checks that they are valid positions in the list (method `checkPosition`) and swaps their elements. Both methods `checkPosition` and `element` may raise runtime exceptions. In total, six methods are analyzed in this example.

[2] `http://net3.datastructures.net`

5.2.10 Sorted Linked List

Program `Sorted Linked List` is a Java implementation of a sorted single-linked list data structure. It was used as a case study in previous work [24]. It consists of a method `add` which adds a `Node` object to the linked list in sorted order. We analyze two subject methods `add3Times` and `add5Times`. These methods invoke method `add` using symbolic integer arguments on an initially empty sorted linked list 3 times and 5 times respectively. This program serves as an example that contains both heap and numeric operations. However, since each of the two methods performs a sequence of method invocations starting from an empty list, the exploration does not build heap path conditions.

5.3 Experimental Results

We have conducted experiments on the subject programs described above using traditional symbolic execution with SPF and our compositional approach. We run symbolic execution several times, once for each method for which we want to build the summary, in a bottom-up fashion. To enable compositional analysis, we added a mechanism for executing each method separately using reflection. The order of methods executed is determined by program's static call graph. We ignore the methods that do not contain any conditional instructions as described in Algorithm 1. For all methods except $MainSymbolic$ no search depth was needed since there is no loops or recursions that drive symbolic execution infinite. For method $MainSymbolic$ in Apollo

to finish in a reasonable amount of time for SPF, we set its search depth as 12, similar to the depth set in [25].

In implementation of our proposed approach we introduced extra choices `BranchChoice`, the search depth in the run of `Comp` was different when the exploration had same program nondeterminism state coverage. To compute the corresponding search depth that has same effects as depth 12 in SPF for Apollo, we counted execution depth when the number of the `non-BranchChoice` along one path accumulated to 12. This computed depth 136 is used in our experiment when executing method `MainSymbolic` in Apollo subject.

Table 5.1 shows the results of our experiments using both traditional SPF (Reg) and our proposed compositional tree-based approach (Comp) for methods in the subjects. We report the number of constraint-solver calls, execution time, number of choices (including `BranchChoice`) explored with SPF, and maximum memory (Mem). We also show the number of paths through the method.

Table 5.1 also shows the tree sizes as the number of nodes in it. "#CTree Node" stands for the number of nodes in a tree that has been compressed, i.e., a path in a caller method is replaced with one *summary node*. "#Tree Node" indicates the number of nodes in the tree before compressing.

The following methods are on leaf nodes in our customized static call graph, i.e., they do not invoke any other methods that contain conditional instructions declared in the subject program so they are analyzed first:

- *deposit* and *withdraw* in BankAccount
- *abs* in Rational
- *update* in WBS
- *main*1 and *main*2 in ASW
- *Non_Crossing_Biased_Descend*
  and *Non_Crossing_Biased_Climb* in TCAS
- *swapNodeWithBoolean* in Swap Node
- *q* in Foo example
- *checkPosition* and *element* in *net.datastructures*

From Table 5.1 we can observe that for these methods the number of constraint solving calls is the same for regular SPF and the compositional approach. This is because building a tree without any existing summary tree cannot reuse any information so the two techniques are basically the same, besides the cost for building and maintaining the extra tree data structure. We can see that for methods *update* in WBS and method *Main*4 in Apollo, building the summary took slightly longer than traditional SPF due to this extra work.

Other methods in Table 5.1 are methods that invoke the leaf methods directly or indirectly. We build summary trees for these methods by reusing

existing summary trees. We can see that Comp incurs fewer number of constraint solver calls and significantly less time than traditional SPF on all these methods. Comp is better for all cases in Table 5.1. In some cases the saving can even be in orders of magnitude. Note that the time listed for these non-leaf methods is the time without building leaf methods summaries. For example, to build summary tree for method *main* in BankAccount, our approach builds summary tree for method *deposit* and *withdraw* first, which took less than 2 seconds in total. Then method summary for *main* is built and it took 6 seconds. So to build a summary tree for method *main* from beginning our approach took 8 seconds, still a better performance compared to traditional SPF's 12 seconds.

Since our implementation introduces the extra *BranchChoice* in SPF, the number of choices reported is much larger than traditional SPF. However, the *BranchChoice* does not introduce any nondeterminism in the state space and thus it does not impose any significant extra work for the analysis. It merely serves as a "dummy" state that records the position and choice of a conditional instruction and introduces extra points for backtracking the search, and so it does not significantly affect the time spent for symbolic execution. The number of these extra choices depend on how many conditional instructions are in the method.

Furthermore, Table 5.1 shows that for every method's execution, our proposed approach uses either the same or less maximum 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 not very meaningful.

Evaluation for error detection. We use five subjects in Table 5.2 to evaluate the effectiveness of our proposed approach with respect to the error treatment. In subject `net.datastructures`, methods `checkPosition` may raise unhandled runtime exceptions in its first branch. For other four subject programs, we inserted an error assertion statement to the methods marked as `build tree` in Table 5.2. For WBS and TCAS we used invalid assertions generated with Daikon from a previous study [24], while for ASW and Apollo we manually inserted errors (assert violations). These assertions are either inserted just before the return instruction or to the last branch of the method.

We used our compositional approach and traditional SPF to symbolically execute these error programs respectively. Table 5.2 shows experimental results for these subjects. For four subjects besides `net.datastructures`, we first built memoization tree for the method that is inserted with error assertions (marked as `build tree`), by reusing the existing method summaries from Table 5.1 as needed. Then we used this updated memoization tree for the modified method to compositionally built the entry method of each subject. During this run, we stopped symbolic execution as soon as we found a path that could lead to an error. The results are shown in lines marked as `comp` in Table 5.2. In comparison, we also ran SPF to symbolically execute the same target entry methods (shown as `SPF`).

From Table 5.2 we can see that our approach speeds up the error de-

tection for both TCAS and WBS. To find the potential exception in method *alt_sep_test* that invoked by method *startTcas*, SPF used 17 seconds while our approach took less than 1 seconds after using 11 seconds to build the summary for *alt_sep_test*. Similarly, in WBS, our approach took 6 seconds to report the first potential error while SPF took 8 seconds. However, for the other three subjects, our approach took longer to build a memoization tree but found the error almost instantly after a summary is acquired.

We conjecture the reason that SPF could find the first error in the program faster than our approach building a tree is that the first error is not "deep" in the program. If the first potential runtime error was in the first few program paths explored by SPF, the execution stops and the rest of paths will not be explored. Unlike SPF, our approach explores all the possible paths in the method during summary construction even if a path can lead to error. Therefore if the first encountered error is in the latter part of the paths that being explored, SPF would take longer to find.

5.4 Threats to Validity

The primary threats to external validity in our study involves the use of SPF for our prototype implementation and the selection of artifacts. However, we attempted to mitigate these threats by analyzing multiple artifacts, most of which have been used in previous studies of symbolic execution based techniques. These threats can be addressed by further evaluation of our technique using a broader range of program types and errors.

The primary threats to internal validity are the potential faults in the implementation of our algorithms and in SPF. We controlled for this threat by testing the tools and implementations of the algorithms on examples that could be manually verified.

With regard to threats to construct validity, the metrics we selected to evaluate the cost of our technique are commonly used to measure the cost of symbolic execution based techniques, but other metrics are possible.

Table 5.1: Experimental Results

Method	# Solver calls		Time(min)		Choices		Mem (MB)		#	#C/Tree
	Reg	Comp	Reg	Comp	Reg	Comp	Reg	Comp	Paths	Nodes
deposit	2	2	<00:01	<00:01	3	5	17	17	2	3/3
withdraw	4	4	<00:01	<00:01	5	10	17	17	3	6/6
main	210	166	0:12	0:06	211	463	17	17	106	169/253

(a) BankAccount

Method	# Solver calls		Time(min)		Choices		Mem (MB)		#	#C/Tree
	Reg	Comp	Reg	Comp	Reg	Comp	Reg	Comp	Paths	Nodes
abs	2	2	<00:01	<00:01	3	5	17	17	2	3/3
gcd	40	40	<00:01	<00:01	25	80	17	17	13	46/46
simplify	92	52	<00:01	<00:01	79	154	17	17	14	28/60
simp	19412	2940	7:13	1:40	16459	37348	48	48	2744	2955/12450

(b) Rational

Method	# Solver calls		Time(min)		Choices		Mem (MB)		#	#C/Tree
	Reg	Comp	Reg	Comp	Reg	Comp	Reg	Comp	Paths	Nodes
update	478	478	0:13	0:22	287	2895	17	17	144	2609/2609
main0	2206	144	0:27	0:03	287	8597	22	20	144	1553/6391
main1	3742	864	0:39	0:09	287	13627	38	28	144	1905/10461
launch	27646	14400	14:00	5:16	27647	305309	200	148	13824	14426/277663

(c) WBS

Method	# Solver calls		Time(min)		Choices		Mem (MB)		#	#C/Tree
	Reg	Comp	Reg	Comp	Reg	Comp	Reg	Comp	Paths	Nodes
main1	42	42	<00:01	<00:01	15	95	17	17	8	81/81
main2	402	402	0:05	0:04	158	1221	17	17	80	640/640
main0	858	576	0:11	0:07	375	5707	17	17	128	457/1809
main	5850	1472	0:52	0:18	375	32907	35	24	128	3104/7313

(d) ASW

Method	# Solver calls		Time(min)		Choices		Mem (MB)		#	#C/Tree
	Reg	Comp	Reg	Comp	Reg	Comp	Reg	Comp	Paths	Nodes
Non_Crossing _Biased_Descend	30	26	<00:01	<00:01	27	73	23	23	14	43/43
Non_Crossing _Biased_Climb	30	26	<00:01	<00:01	27	73	23	23	14	43/43
alt_sep_test	678	230	0:20	0:07	135	1579	25	23	68	264/901
startTcas	2348	100	0:59	0:04	135	5238	39	29	84	361/2682

(e) TCAS

Method	# Solver calls		Time(min)		Choices		Mem (MB)		#	#C/Tree
	Reg	Comp	Reg	Comp	Reg	Comp	Reg	Comp	Paths	Nodes
Main4	12	12	0:01	0:02	8	75	17	17	4	55/55
Main5	12	12	0:01	0:01	4	47	17	17	3	43/43
Main6	12	12	0:01	0:01	9	94	17	17	4	59/59
Main1	348	279	2:43	2:08	153	4788	18	17	31	179/1106
MainSymbolic	2256	1089	35:00	16:12	423	29236	18	18	175	688/2270

(f) Apollo

Method	# Solver calls		Time(min)		Choices		Mem (MB)		#	#C/Tree
	Reg	Comp	Reg	Comp	Reg	Comp	Reg	Comp	Paths	Nodes
swapNode WithBoolean	16	16	<00:01	<00:01	20	39	16	17	6	13/13
callTwice	742	100	0:06	0:01	1556	2174	17	17	36	43/85
call3Times	48056	4268	12:29	1:10	96963	133461	58	24	216	259/517

(g) SwapNodeWithBoolean

Method	# Solver calls		Time(min)		Choices		Mem (MB)		#	#C/Tree
	Reg	Comp	Reg	Comp	Reg	Comp	Reg	Comp	Paths	Nodes
q	236	236	0:01	0:01	346	698	15	15	11	21/21
m	255976	21903	22:09	10:37	256886	711261	15	15	125	158/338

(h) Foo

Method	# Solver calls		Time(min)		Choices		Mem (MB)		#	#C/Tree
	Reg	Comp	Reg	Comp	Reg	Comp	Reg	Comp	Paths	Nodes
checkPostion	0	0	<00:01	<00:01	13	27	16	16	4	9/9
element	0	0	<00:01	<00:01	23	31	16	16	4	7/7
swapElements	0	0	00:01	00:01	5875	7341	17	27	7	13/20

(i) Net Data Structure

Method	# Solver calls		Time(min)		Choices		Mem (MB)		#	#C/Tree
	Reg	Comp	Reg	Comp	Reg	Comp	Reg	Comp	Paths	Nodes
add	4	4	<00:01	<00:01	5	14	17	17	3	10/10
add3Times	10	8	<00:01	<00:01	11	31	17	17	6	10/25
add5Times	238	86	00:06	00:03	239	749	16	16	120	308/511

(j) Sorted Link List

Table 5.2: Experimental results for error treatment

Method	# Solver calls	Time(min)	States	Mem (MB)
TCAS.alt_sep_test(build tree)	312	0:11	3032	20
TCAS.startTcas(comp)	0	<00:01	1	20
TCAS.startTcas(SPF)	652	0:17	39	27
WBS.update(build tree)	526	0:05	2979	16
WBS.Main1(comp)	0	<00:01	1	16
WBS.Main1(SPF)	1142	0:08	175	16
ASW.main0(build tree)	576	0:07	8331	17
ASW.main(comp)	0	<00:01	1	39
ASW.main(SPF)	26	<00:01	8	17
Apollo.Main1(build tree)	391	2:36	2331	23
Apollo.Main1(comp)	0	<00:01	1	20
Apollo.MainSymbolic(SPF)	39	0:27	5	18
Net.datastructures.swapElements(comp)	0	<00:01	1	16
Net.datastructures.swapElements(SPF)	0	<00:01	2	16

Chapter 6

Related Work

A number of papers have addressed compositional symbolic execution [3, 4, 10, 11]. Also compositional techniques with inter-procedure analysis have been studied extensively in [6, 9, 12].

Godefroid [10] is the first one to propose compositional techniques to improve automated test input generation using "dynamic" symbolic execution, i.e., symbolic execution performed along a particular concrete path. Based on it, Anand et al. [4] present a (demand-driver) top-down compositional symbolic execution in the context of dynamic test generation. Their approach utilizes execution trees similar to ours however they are not used for replay. Instead, their method summary is a first-order logic formula with uninterpreted functions and the compositional symbolic execution is performed entirely using SMT solving. These works do not address composition in the presence of heap operations for object-oriented programming.

Albert et al. [3] address the construction of method summaries in the presence of heap operations. Their method summaries include both logical formulae and an explicit representation of the input and output heaps. Thus the summary encodes all the effects of the computation so once a summary

is acquired no "replay" is needed. However, such advantage comes at a high price—method summaries are large and the composition operation is complicated. This is because it not only checks compatibility at the invocation site, but also synthesizes new state and new heap to continue with after method composition. The composition operation is performed in the context of constraint logic programming (CLP), which can not be employed efficiently in a general purpose tool such as SPF.

Rojas et al. [19] have described a preliminary investigation of compositional symbolic execution for Java bytecode analysis. Their approach builds a method summary that contains a set of summary cases, each case consisting of a "path-specialized" version of the method code. Summary case is generated by using partial evaluation, a well-established technique that aims at automatically specializing a program with respect to some of its input. The benefits obtained with that approach are not very impressive since storing multiple versions of the code is expensive. In contrast, our proposed approach uses much lighter weight method summaries that only encode the choices taken along each path. This representation is sufficient for method replay. Nevertheless, the ideas from [19] served as inspiration for this thesis.

Another recent related technique to ours is presented by Cho et al. [7], although that work is done in the context of bounded model checking not symbolic execution, and it performs weakest preconditions calculations and not forward computations. Furthermore it does not handle heap operations but it is tailored towards the properties it aims to check.

Yang et al. [25] propose "memoized" symbolic execution for efficient re-application of symbolic execution in different scenarios: with iterative deepening of exploration depth, to perform regression analysis, or to enhance coverage using heuristics. Memoized symbolic execution uses similar tree data structure to encode key elements of symbolic execution and stores it (on disk) for reuse. However, their tree data structure is not compositional and the symbolic execution of the whole program is stored in a single tree. The proposed approach in this thesis naturally extends their tree data representation to a finer granularity, i.e., at the level of procedures, resulting in increased efficiency.

Chapter 7

Conclusion

This thesis introduced 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 an efficient treatment of error traces by short-circuiting the execution of paths that lead to them. Our preliminary experimental evaluation based on a prototype implementation in Symbolic PathFinder showed 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.

Bibliography

[1] SMT-COMP 2011. http://www.smtcomp.org/2011/.

[2] SIR Repository. http://sir.unl.edu.

[3] Elvira Albert, Miguel Gmez-Zamalloa, Jos Miguel Rojas, and Germn Puebla. Compositional clp-based test data generation for imperative languages. In *LOPSTR'11*, 2011.

[4] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In *TACAS*, pages 367–381, 2008.

[5] David Brumley, Juan Caballero, Zhenkai Liang, James Newsome, and Dawn Song. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *SS*, pages 15:1–15:16, Berkeley, CA, USA, 2007. USENIX Association.

[6] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *POPL'09*, 2009.

[7] Chia Yuan Cho, Vijay D'Silva, and Dawn Song. Blitz: Compositional bounded model checking for real-world programs. In *ASE*, pages 136–146, 2013.

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

[9] Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In *PLDI'11*, 2011.

[10] Patrice Godefroid. Compositional dynamic test generation. In *POPL*, pages 47–54, 2007.

[11] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and Sai Deep Tetali. Compositional may-must program analysis: unleashing the power of alternation. In *POPL*, pages 43–56, 2010.

[12] Susan Horwitz, Thomas W. Reps, and Shmuel Sagiv. Demand interprocedural dataflow analysis. In *SIGSOFT FSE*, pages 104–115, 1995.

[13] Kobi Inkumsah and Tao Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *ASE*, pages 297–306, 2008.

[14] A. Joshi and M. P. E. Heimdahl. Model-Based Safety Analysis of Simulink Models Using SCADE Design Verifier. In *SAFECOMP*, volume 3688 of *LNCS*, pages 122–135, September 2005.

[15] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *TACAS*, pages 553–568, 2003.

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

[17] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. Directed incremental symbolic execution. In *PLDI*, pages 504–515, 2011.

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

[19] Jos Miguel Rojas and Corina S. Pasareanu. Compositional symbolic execution through program specialization. In *BYTECODE'13 (ETAPS)*, 2013.

[20] SAE-ARP4761. *Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*. SAE International, December 1996.

[21] Matheus Souza, Mateus Borges, Marcelo d'Amorim, and Corina S. Păsăreanu. CORAL: solving complex constraints for Symbolic PathFinder. In *NFM*, pages 359–374, 2011.

[22] Janos Sztipanovits and Gabor Karsai. Generative programming for embedded systems. In *GPCE*, pages 32–49, 2002.

[23] Michael W. Whalen, Patrice Godefroid, Leonardo Mariani, Andrea Polini, Nikolai Tillmann, and Willem Visser. Fite: future integrated testing environment. In *FoSER*, pages 401–406, 2010.

[24] Guowei Yang, Sarfraz Khurshid, Suzette Person, and Neha Rungta. Property differencing for incremental checking, to appear in ICSE 2014.

[25] Guowei Yang, Corina S. Pasareanu, and Sarfraz Khurshid. Memoized symbolic execution. In *ISSTA*, pages 144–154, 2012.