Compositional Symbolic Execution using Fine-Grained Summaries

Yude Lin

Computing and Information Systems
The University of Melbourne
Victoria 3010, Australia
Email: yudel@student.unimelb.edu.au

Tim Miller

Computing and Information Systems
The University of Melbourne
Victoria 3010, Australia
Email: tmiller@unimelb.edu.au

Harald Søndergaard

Computing and Information Systems
The University of Melbourne
Victoria 3010, Australia
Email: harald@unimelb.edu.au

Abstract—Compositional symbolic execution has been proposed as a way to increase the efficiency of symbolic execution. Essentially, when a function is symbolically executed, a *summary* of the path that was executed is stored. This summary records the precondition and postcondition of the path, and on subsequent calls that satisfy that precondition, the corresponding postcondition can be returned instead of executing the function again. However, using functions as the unit of summarisation leaves the symbolic execution tool at the mercy of a program designer, essentially resulting in an arbitrary summarisation strategy. In this paper, we explore the use of *fine-grained* summaries, in which blocks *within* functions are summarised. We propose three types of summarisation and demonstrate how to generate these. At such a fine-grained level, symbolic execution of a path effectively becomes the concatenation of the summaries along that path. Using a prototype symbolic execution tool, we perform a preliminary experimental evaluation of our summary approaches, demonstrating that they can improve the speed of symbolic execution by reducing the number of calls sent to the underlying constraint solver.

I. INTRODUCTION

Path explosion is one of the toughest obstacles for symbolic execution. The number of potential paths in a program’s execution tree grows exponentially with the number of branching statements (including loops), creating an infeasibly large search space for symbolic execution tools. This significantly compromises performance and limits symbolic execution’s range of application in real-world software testing.

Compositional Symbolic Execution (CSE) [1], [2], [3] is one strategy to help alleviate path explosion. CSE reuses previously generated path constraints to reduce the pressure on path search, constraint generation and most importantly, constraint solving. It can intelligently eliminate redundant constraint solving, and can be embedded into other types of symbolic execution (dynamic, for instance). Empirical studies on CSE [1], [2], [3] have demonstrated that using *function summaries* — summaries of entire paths through functions — can reduce the number of constraints and hence execution time for symbolic execution. A summary consists of a precondition and postcondition of a single path through a function. When a symbolic execution tool encounters a function call, it first looks to see if the corresponding constraint satisfies the precondition of a path that has been executed previously. If so, it updates the symbolic state using the postcondition, instead of executing the function again.

In this paper, we investigate the use of *fine-grained* summaries in symbolic execution; that is, the use of summarising blocks of code *within* functions. Summaries are generated by collecting *condition pairs*, which describe the *entry condition* (the weakest precondition) of a code fragment, and the *postcondition* achieved under that weakest precondition. Summarising at a finer grain provides more opportunities for re-use of summaries, thus our hypothesis is that fine-grained summaries will reduce the number of calls to a constraint solver. If summaries are produced for all blocks, path constraints can be completely built by summaries. The result is that the execution can be forwards or backwards.

Summaries are used for fast referencing of generated constraints, saving time when exploring identical subtrees. Consider the program snapshot in Figure 1 and its loop body control flow graph in Figure 2. Conventional symbolic execution unfolds the loop into consecutive copies of the loop body (from line 2 to line 10) guarded by the loop condition, sharing the same subtree structure. In the subtree structure, there are in total four possible routes (one of which is infeasible) and three forking points generated from the nested *if* statements. Hence six constraint solver calls (two for each fork) are made each time the search deepens into a loop body copy, with one call known to return failure each time.

A useful summarisation in this case spans line 2 to line 10. This summarisation focuses on the corresponding subtree and summarises it into 3 subpath constraints (the summaries), discarding the infeasible subpath. Each time the search encounters the loop body, these summaries can be *concatenated* to form new path constraints. After three subpath constraints are cached as summaries, only the blue calls (a, b,

```

while (x > 0) {
    if (a[x] != 0) {
        c++;
        if (b[x] == a[x]) {
            d++;
            if (b[x] == 0)
                e++;           // dead code
        }
    }
    x--;
}

```

Fig. 1. A loop with a multi-path body.

we are thus able to assemble a new path constraint using parts taken from the other ones if they share *any* instruction sequence. The parts that are kept for reuse are the *summaries*.

B. Condition Pairs

CSE generates path constraints by putting together constraints of the path components. In our approach, the smallest component of a path that we can collect constraint information from is a single instruction. The idea of summary is to use *entry conditions* and *postconditions* to describe any instruction I or sequence of instructions S :

Definition 1: A *summary* on an executable instruction sequence S provides the conditions under which S is executed from the beginning to the end without a control flow escape, as well as the consequences of successfully executing S . It can be represented by a *condition pair*:

$$CP(S) = \left\{ \begin{array}{l} EC(S) \\ PC(S) \end{array} \right\}.$$

The *entry condition* $EC(S)$ is the weakest proposition under which S is executed. The *postcondition* $PC(S)$ is the strongest proposition about the state that results from execution of S under its entry condition.

Hence if S consists only of a basic block of instruction(s), such as assignment(s), then $EC(S) = \text{true}$, indicating that S will be executed to its end, no matter what. The entry condition for a sequence is only constrained by conditional jumps (such as branch and loop guards). By definition, if S is a sequence of instructions forming into a program path, then $EC(S)$ is the path constraint. Note that $EC(S)$ is always implied by $PC(S)$, assuming the variables are named with consistent version numbers. In other words, in a condition pair, the postcondition is always at least as strict as the entry condition.

For each individual instruction other than a conditional branch, its condition pair is generated before summarisation:

$$\begin{aligned} I = op_2 v_x, v_y &\mapsto \begin{array}{l} EC(I) = \text{true} \\ PC(I) = (v_{x_1} = v_{y_0}) \end{array} \\ I = op_3 v_x, v_y, v_z &\mapsto \begin{array}{l} EC(I) = \text{true} \\ PC(I) = (v_{x_1} = v_{y_0} \wedge v_{z_0}) \end{array} \\ I = control &\mapsto \begin{array}{l} EC(I) = \text{true} \\ PC(I) = \text{true} \end{array} \end{aligned}$$

A conditional branching instruction has no condition pair, but gets one during concatenation. For example, if I is the instruction `branch_on_false label` and I is followed by an instruction sequence S_1 ; $label : S_2$, then I will get one of two condition pairs the moment it gets concatenated, according to the sequence of code it is concatenated with. The resulting condition pair will look like one of these:

$$\begin{aligned} CP(I : S_1) &= \left\{ \begin{array}{l} r0_0 = \text{true} \\ r0_0 = \text{true} \end{array} \right\} \otimes CP(S_1) \\ CP(I : S_2) &= \left\{ \begin{array}{l} r0_0 = \text{false} \\ r0_0 = \text{false} \end{array} \right\} \otimes CP(S_2) \end{aligned}$$

The $\otimes$ symbol is the concatenation operator, described in Section II-C. As mentioned, `r0` is the default location for Boolean results, which can be from a preceding comparison. The other conditional branching instruction will follow a similar rule.

```

int_const r0, 0
read r1      # read input
cmp_gt_int r1, r0
branch_on_false fb
str_const r0, "T"
store slot0, r0
branch_uncond next
fb: str_const r0, "F"
store slot0, r0
next: ..

```

Fig. 3. Example code fragment.

For larger part of the code, we first have the concept of code fragment defined below:

Definition 2: A *code fragment* is a connected portion of a program, which has a CFG G_f that is a subgraph of the program's CFG G_p , where G_f has entry nodes $EntN_i$ where $i \in 1..n$ and exit nodes $ExiN_j$ where $j \in 1..m$. Each path in this fragment—a subpath of the program—starts with a node in $EntN_i$ and ends with a node in $ExiN_j$. In IR form, a code fragment contains one or more executable instruction sequences, each of which implements one of the fragment's paths.

Note that in the specific context of our backward reasoning implementation described in Section IV, we have $n = 1$.

As an example, in Figure 3 is an if-then-else statement in IR form. The code fragment has two paths, one for each branch. The true branch involves the instruction sequence $S_t = 1-2-3-4-5-6-7$, and the false branch $S_f = 1-2-3-4-8-9$. The condition pairs of the two paths will be:

$$\begin{aligned} EC(S_t) &= (r0_1 = 0 \wedge r1_1 > r0_1) \\ PC(S_t) &= (r0_1 = 0 \wedge r1_1 > r0_1 \wedge \\ &\quad r0_2 = \text{"T"} \wedge slot0_1 = r0_2) \\ EC(S_f) &= (r0_1 = 0 \wedge r1_1 \leq r0_1) \\ PC(S_f) &= (r0_1 = 0 \wedge r1_1 \leq r0_1 \wedge \\ &\quad r0_2 = \text{"F"} \wedge slot0_1 = r0_2) \end{aligned}$$

Note that here we assume we are doing forward reasoning, and the latest version of all variables before this code fragment is 0, meaning that the condition pairs' variables' base version is all 0. Each time a variable is assigned a value, its version is incremented by 1. The derivation of such complex condition pairs from individual condition pairs uses the *concatenation* operation.

C. Concatenation

Concatenation is an operation on two condition pairs, each of which could be concatenated condition pairs themselves. Systematic concatenation allows us to construct complex condition pairs from individual condition pairs, and eventually obtaining the path constraint for a given program path. The concatenation rule is

$$CP(S_1) \otimes CP(S_2) \mapsto CP(S_1; S_2)$$

where

$$\begin{aligned} EC(S_1; S_2) &= EC(S_1) \wedge (EC(S_2) \mid PC(S_1)) \\ PC(S_1; S_2) &= PC(S_1) \wedge PC(S_2) \end{aligned}$$

$S_1; S_2$ represents the sequence of S_1 followed by S_2 . The *connect condition* $(EC(S_2) \mid PC(S_1))$ is, given $PC(S_1)$, the weakest condition to make $EC(S_2)$ true. More precisely, let $V = vars(PC(S_1)) \setminus vars(EC(S_2))$. Then

$$(EC(S_2) \mid PC(S_1)) \equiv \forall V (PC(S_1) \Rightarrow EC(S_2)).$$

The connect condition is simply suggesting that we hide all of the intermediate variables during concatenation (through variable matching and substitution). For example, $(y < 0 \mid x = 1)$ is equivalent to $y < 0$ and $(y < x \mid x = 1)$ is equivalent to $y < 1$. Alternatively, we can use

$$EC'(S_1; S_2) = PC(S_1) \wedge EC(S_2)$$

as the entry condition. It results in longer constraints, but is logically equivalent to $EC(S_1; S_2)$ regarding the possible values of variables in $EC(S_1; S_2)$. Only intermediate or irrelevant variables are constrained in the additional constraints, so that the feasible models can always be found.

Recall that in S_1 and S_2 , there are different version number assignments for the variables with the same names. Version numbers are updated to preserve consistency before concatenation. At each concatenation, variables are replaced with fresh variable names, using version numbers in ascending order for readability; but any fresh variable name will suffice.

D. De-concatenation

De-concatenation is the inverse operation of concatenation. We use de-concatenation to remove newly concatenated condition pairs from a bigger condition pair so that the search can backtrack to a previous branching point, and so we do not have to cache summaries of all subpaths.

If we use the second entry condition representation (EC'), the corresponding de-concatenation will be simply cutting off a part of the given condition pair. If we use the first representation (EC), where there are connect conditions, we have to search in the postcondition in the given condition pair, to recover the original $EC(S_2)$, then we can do the cutting.

Currently we have only implemented very simple connect conditions. The consequences are that we have longer constraints, but they are all repeated or intermediate constraints, which means they are very easy for the solver to reduce and solve. Besides the de-concatenation is faster. We use the simplest way to describe entry conditions in the following paragraphs and do not distinguish between EC and EC' .

E. Composing path constraints

A path constraint is simply the concatenation of the condition pairs from all code fragments in that path. Essentially our CSE approach is a process of finding the possible sequences of instructions, concatenating the condition pairs to achieve the path constraint, and solving this path constraint.

Table I shows how two path constraints can be built to the code fragment in Figure 3, assuming a forward reasoning mechanism.

We omit some of the variables (*input*, *temp*) and the last few steps for simplicity. Also we assume that the end of a branch is the end of the search and we backtrack immediately. Each forward step contains two operations: version number updating and concatenation, using the rules previously defined. There is a special instruction providing an indefinite condition pair, the conditional branching instruction. Symbolic execution marks its location as a backtracking point. When going through this point, the symbolic execution will generate a condition pair in accordance with the selected branch.

The example assumes summaries are at the instruction level. If there are summaries of larger blocks available (we record mappings to indicate from where to where summaries are available) in the concurrent path, symbolic execution will use the summaries instead of instruction condition pairs, skipping what is between the starting point and the end point(s) of the summaries. This is how the summarisation mode of symbolic execution avoids redundant solver calls. Again, summaries are essentially cached condition pairs.

F. Direction of SSE

SSE can be categorised as forward reasoning and backward reasoning, according to the manner in which it searches for feasible paths and collects constraint information. In general, backward reasoning is not reacting as quickly to an infeasible path as forward reasoning. Forward reasoning makes decisions to discard a branch choice by looking at the conditions at the beginning of the branch, while backward reasoning looks for useful information in the branch. Moreover, some useful information is not made available to a backward approach in time, e.g., initialisation of variables.

When doing directed symbolic execution — that is, symbolic execution towards a specific statement — the conditions near the search target are not made available in time to a forward approach. In this case, a backward approach could be superior because it starts with this information.

Our summarisation technique can be flexibly combined with both reasoning methods. The order of reasoning about the instructions in a program depends on the order of summary concatenation.

Over the past decade, the combination of symbolic execution and concrete execution has made forward reasoning much easier [8], [9], [10], [11], [12], [13], but concrete execution's nature does not allow reasoning backwards. As such, BSE is much more vulnerable to path explosion than forward symbolic execution. Our CSE tool (discussed in Section IV) uses backward reasoning, but this is not to suggest that fine-grained summarisation has less potential when combined with forward reasoning.

III. USING SUMMARIES

Summaries are the building blocks of path constraints in compositional symbolic execution. Existing summarisation techniques include partial loop summarisation [14], whose summary generation is based on loop and loop invariant detection heuristics, which is a different topic (see Section VI); and function-level summarisation [1], [2], [3], meaning that the size of the target summaries are chosen based on design

TABLE I. BUILDING THE TWO CONDITION PAIRS OF THE EXAMPLE IN FIGURE 3 STEP BY STEP.

Current sequence of instructions	Current condition pair	Next instruction	Next condition pair
NULL	<i>EC</i> <i>true</i> <i>PC</i> <i>true</i>	1	<i>true</i> $r0_1 = 0$
1	<i>EC</i> <i>true</i> <i>PC</i> $r0_1 = 0$	2	<i>true</i> $r1_1 = input$
1-2	<i>EC</i> <i>true</i> <i>PC</i> $r0_1 = 0 \wedge r1_1 = input$	3	<i>true</i> $(r1_0 > r0_0) = temp$
1-2-3	<i>EC</i> <i>true</i> <i>PC</i> $r0_1 = 0 \wedge (r1_1 > r0_1) = temp$	4	<i>temp</i> = <i>true</i> <i>temp</i> = <i>true</i>
1-2-3-4	<i>EC</i> $r0_1 = 0 \wedge r1_1 > r0_1$ <i>PC</i> $r0_1 = 0 \wedge r1_1 > r0_1$	5	<i>true</i> $r0_1 = "T"$
1-2-3-4-5-6-7	.. (Reach the end of branch and backtrack)		
1-2-3	<i>EC</i> <i>true</i> <i>PC</i> $r0_1 = 0 \wedge (r1_1 > r0_1) = temp$	4	<i>temp</i> = <i>false</i> <i>temp</i> = <i>false</i>
1-2-3-4	<i>EC</i> $r0_1 = 0 \wedge r1_1 \leq r0_1$ <i>PC</i> $r0_1 = 0 \wedge r1_1 \leq r0_1$	8	<i>true</i> $r0_1 = "F"$
1-2-3-4-8-9	.. (Reach the end of branch)		

choices made by a developer. As a consequence, the function summarisation is still vulnerable to path explosion inside functions. In this section, we explore the appropriate level of summarisation from within functions.

Our target of summarisation can range from one single instruction to multiple basic blocks, including entire paths. We describe three strategies of choosing good summarisation targets later in this section. Before that, we discuss the principles of identifying desirable blocks, which also clarifies why summarisation on a large code fragment; e.g., at a function level, is not always desirable.

A. Program Characteristics and Summarisation

The major cost of symbolic execution comes from constraint solving. One of the benefits of using summaries is the elimination of redundant solver calls. However, summaries come with certain costs, including the sacrifice of some of the decision making points and the cost of computing and storing summaries themselves.

In the example in Figure 2, the decision making points in summarisation mode are postponed to the end of the subpaths. Suppose “call 2” is always unsatisfiable for some reason (maybe there are other conditions constraining array *a* outside the loop), of which summarisation would not know because summarisation is done locally. Every time the search enters the loop body, three summaries will be used while only one of them is of value (the one with a blue “call a”), the other two are destined to fail. This inefficiency exposes two problems of summarisation: 1) latency in decision making: if we execute “call 2”, the satisfiability would have been checked earlier; and 2) target sensibility: the perfect code fragment to be summarised should not have too many variables constrained elsewhere. The longer the summaries are, the larger delay there could be. For this reason, long summaries, such as function-level summaries, are not as desirable as shorter code-fragment summaries.

We now discuss desirable features of candidate code fragments to be chosen.

1) Number of Infeasible Subpaths: A subpath is infeasible when its path constraint is unsatisfiable. A naive path search will try such a path as long as it is topologically possible,

or until the constraint solver stops it from doing so. In other words, an infeasible subpath will consume solver calls while it does not provide the symbolic execution with a useful result. Eliminating these from consideration can reduce the number of calls sent to a solver.

Our fine-grained summarisation technique can identify a subset of the infeasible subpaths in the summarised code. When a subpath is shown infeasible, i.e., we have generated an unsatisfiable entry condition for this subpath, its condition pair will be not be cached, so throughout the remaining symbolic execution, the infeasible subpath will no longer be considered.

Rule number one for summarisation would be to look for code fragments that contain infeasible subpaths. The remaining infeasible subpaths that are not detected are potential “threats” to summarisation. We discuss this problem with the next code characteristic.

2) Independent Code: A code fragment has lower priority in summarisation if it has a strong dependency on other parts of the program. Strong dependency means that the variables in this code fragment are partly constrained elsewhere. In this case, the summarisation procedure is unlikely to discover infeasible subpaths due to the lack of constraint information. More importantly, the symbolic execution runs a higher risk of generating long summaries, which may be unsatisfiable.

The reason is as follows. Our summarisation is a local analysis on a target code fragment, and does not take into consideration any surrounding blocks, so as to allow the resulting summaries to fit into any environment. Hence in a summary, there may exist weak constraints that are only unsatisfiable provided the surrounding context. Such weak constraints means the summarisation can produce more summaries (for each potential subpaths) than there actually needs to be. For example, a variable may be initialised to a particular value when declared. If a subsequent block that does not include the initialisation statement is summarised, the summary will over-approximate the possible execution subpaths. The should-be-eliminated summaries could be added during path search and remain undetected until the right intialisation information is given. The longer the summarisation target is, the more such summaries there will be and the more constraint solving time they will take.

Rule number two for summarisation would be to avoid breaking a functional unit apart for summarisation; instead, independent code fragments should be summarised. For example, loops in drivers listening to user actions and performing independent tasks in each of their iterations [15] are good targets for summarisation. An isolated function that does not use global variables and whose return value is not strongly constrained are also suitable.

3) *Minimum Number of Calls Ratio*: Combining the two considerations, one should summarise a code fragment when it can be expected to contain infeasible subpaths. If the code fragment is rather independent, that is, the satisfiability of the subpaths is not depending on information given outside the fragment, then summarisation is preferable. The more frequently it is visited, the more effective the summaries will be.

Summarisation tries to reduce the number of calls needed for search in the selected code fragment. The ratio

$$R_c = \frac{\text{No. of all resulting summaries}}{\text{No. of calls to summarise the code fragment}}$$

expresses the efficiency of the summarisation on a code fragment. The efficiency is higher when R_c is smaller. In Figure 2, during summarisation (local symbolic execution in the loop body), 6 calls are made in advance to create 3 subpath summaries for symbolic execution. The summarisation would expect the symbolic execution to make use of those 3 subpath summaries instead of 6 calls every time in the search, so that $R_c = \frac{1}{2}$. Actual efficiency could be different due to a possible dependency problem, as we explain in Section III-A2, in which case a non-summarisation symbolic execution tool can use only 2 calls to explore the code instead of the predicted 6, and the summarisation is less effective.

R_c reflects whether a summarisation strategy has considered the program characteristics. If a code fragment has a large portion of its paths infeasible and they can be detected locally, R_c will be small. Otherwise it can be close to 1, meaning summaries from this fragment are not so valuable. We set a threshold on R_c and check against it each time we finish summarisation on one code fragment to prevent weak or useless summaries (implementation explained in Section IV-A). A low R_c can guarantee increase in symbolic execution speed in most cases, witness the results in Section V.

B. Strategies for Summarisation

In this section, we present three heuristics for finding good summarisation targets. As previously explained, in general, the target should be a repeatedly visited code fragment. The three heuristics differ in the degree of conservatism. Their features and expected efficiency, relative to each other, are stated qualitatively in Table II. Note that summarising a larger code fragment tends to yield more, and longer, summaries because of the larger size of execution subtree. The “negative impact of dependencies” predicts how much the real efficiency departs from the expected efficiency when variables inside and outside the summarised code fragments are strongly related.

Summarisation is performed before symbolic execution, which means the conditional statement detection is through a recovered CFG. All summaries are cached, as well as their starting and ending location.

TABLE II. SUMMARISATION STRATEGIES.

Strategy	LSUM	LASUM	SSUM
Speciality	Loops	Loops	Consecutive ite statements
Nested summarisation	Yes	Yes	Yes
Expected length of summaries	High	Medium	Low
Expected number of summaries	High	Medium	Low
Negative impact of dependencies	High	Medium	Low

1) *Loop-Body (LSUM)*: In this summarisation mode, any loop body including the loop condition is summarised. If there is a nested inner loop, this inner loop will be summarised first, and then the summaries of the nested loop will be used to construct an enclosing loop’s summaries. Each summary in this mode represents the subpath constraint of each subpath going through the loop body in one loop iteration.

This heuristic suits a program with loops whose iterations are rather independent. Loops manipulating arrays or lists, with each iteration not interfering with others, are ideal targets. For a program with a large nested loop, this mode may be prohibitively expensive, owing to possible path explosion inside the loop.

2) *Acyclic-Loop-Body (LASUM)*: In this mode, similarly, loop bodies will be summarised. The difference is, inner nested loops break the outer loop’s summarisation. As above, the inner loop is summarised first, and then, the contents before and after the inner loop, but inside the outer loop, are summarised respectively.

This heuristic also works well if there is independence among the iterations. For nested loops it mitigates the path explosion problem seen with LSUM, but is more conservative (produces few summaries), lowering the potential to reduce solver calls.

3) *Small-Step (SSUM)*: In this mode, successive if-then-else statements and interleaved or adjacent basic blocks in the same scope are summarised, provided none of them contain a loop. Statements at different levels of nesting are summarised separately, with more deeply nested conditionals summarised first. However, the number of if-then-else statements summarised each time is limited by a threshold value.

This heuristic produces smaller summaries than the previous two heuristics and is inexpensive to calculate. Even when the summaries cannot reduce the symbolic execution’s solver call need significantly, there is minimal disadvantage.

4) *Discussion*: Figure 4 provides a general outline of the three strategies on a schematic program. One can see that the difference between the three heuristics is how they restrict the area of summarisation. The order of strictness is generally $SSUM > LASUM > LSUM$.

Under any summarisation mode (XSUM), the innermost scope will be summarised first using XSUM. The larger scope is then X-summarised using the previously generated summaries. In this way, path explosion in a large target is better controlled. Theoretically, we can interchangeably use different summarisation modes on the same code fragment, which will be of help when the a single code fragment is large enough; however in our experiments later, we assess the strategies individually.

Fig. 4. Illustration of how different summarisation strategies selectively summarise a program (the interesting part, a loop). “SSUM_3” means in this case the limit of the number of consecutive if-then-else statements is 3 in SSUM mode.

IV. IMPLEMENTATION

Our tool is an experimental prototype, not necessarily intended to scale to large systems. Nonetheless, it is sophisticated enough to experiment with summarisation ideas on non-trivial programs, as described in the evaluation (Section V). The tool accepts programs written in a basic C-like language with assignment, branching, and looping. The types supported include integers, floats, and arrays of these. Strings can be used but not in constraints. This is compiled to an intermediate assembler-like language, upon which we apply our algorithms. We use Z3 [16] as the underlying constraint solver, to both produce summaries and perform the symbolic execution. Incremental solving is turned on, and constraints on the path constraint are pushed and popped in accordance with the search algorithm.

A. Summarisation and the R_c Threshold

Summarisation is performed before symbolic execution. It is essentially a small symbolic execution done on the code fragments automatically selected by the summarisation strategy. The results are the corresponding summaries for each of these code fragments in the form of condition pairs.

After that the summarisation needs to go through a screening process for the reason stated in Section III-A3. A summarised code fragment needs to satisfy

$$R_c < \text{threshold}(\text{threshold} \in (0, 1))$$

for its summaries to be of any use. Otherwise we discard the summaries and choose to use the naive path search later in the symbolic execution, because a summarisation above the threshold gives the smallest boost up in the speed while having a higher risk of containing dependent constraints. The smaller the *threshold* is, the fewer summaries we will have, which means we turn down the intensity of summarisation to gain robustness.

B. Search Strategies

The symbolic execution performs path search as follows. The starting point is the target instruction. First, it finds all

instructions that directly precede the current instruction in the CFG, and chooses one of them to proceed according to the specified search strategy (discussed below). Then it concatenates the “current” condition pair and the newly found condition pair. The entry condition in the concatenated condition pair will be pushed to the solver’s stack and checked. Any summary found in the middle of this procedure will be used instead of individual instructions’ condition pairs, and the search will skip to the beginning instruction of the selected summary. If a program entry is found, then the search has ended and it will backtrack until there is no more choice.

The summarisation allows more search heuristics because of enlarged search steps. At each branching point where summaries are available, different priority schemes can apply. Summaries gather information from a larger range of code, which encourages the decision maker to make a wiser choice while doing targeted symbolic execution. Since in BSE the target is always the program entry (in our case there is only one entry with a certain instruction number 0), we have only implemented one search strategy—greedy search. However, when doing forward targeted symbolic execution, we believe there could be more possibilities—this will be one direction of our future work.

At each branching point, the greedy search strategy will choose a previously-unchosen preceding instruction/summary that has the smallest instruction number/starting point number, in order to get closer to the program entry. To avoid getting stuck in a search, each path considered has a parameterisable maximum depth.

V. PRELIMINARY EVALUATION

In this section, we present an evaluation of our fine-grained-summarisation-based symbolic execution. We compare the three strategies, LSUM, LASUM, and SSUM, alongside an approach that uses no summarisation, whose search pattern is effectively equivalent to standard BSE. We measure the execution time and the number of calls to the constraint solver.

A. Experiment Design

TABLE III. OBJECTS OF ANALYSIS.

Program	# Instrs	# Conds	# Loops
binary_search	117	5	2
bubble_sort	190	3	2
cba_example	125	6	3
due_date	432	6	1
kmp	408	8	5
matrices	518	8	7
tic_tac_toe	951	18	1
wumpus	1,823	270	12

At this stage our prototype symbolic execution tool accepts a simple subset of C, providing us with a rather limited pool of test programs. Nonetheless it is sufficient for the observation of our summarisation technique’s performance. We take eight small but non-trivial programs, summarised in Table III. The “Conds” are the number of branches, some of which represent loops. Although these programs are small in terms of instructions, they still have an explosively large amount of paths; in some cases, infinite.

`binary_search` and `bubble_sort` are the well-known searching and sorting algorithms. `cba_example` is the example program taken from Figure 1(a) of Obdržálek and Trtík [17]. `kmp` is the Knuth-Morris-Pratt algorithm.

`due_date` computes a due date from a start date and a period (in number of days). `matrices` is a matrix multiplication program. These two programs contain multiplication and/or division operations, which are comparably hard for the solver, and this is reflected later in the results (see Table IV). `tic_tac_toe` and `wumpus` are programs of the games *Tic-Tac-Toe* and *Hunt the Wumpus* respectively.

We varied one independent variable in our experiment: the summarisation mechanism used. This implies four different values: (1) CSE: only individual instructions' condition pairs are cached; (2) LSUM: the simple loop summarisation approach; (3) LASUM: the acyclic loop summarisation approach; and (4) SSUM: the small-step summarisation approach, with the length limit set to three. The threshold of R_c is set to 0.8 for all the summarisation modes.

We measured the execution time for each approach on each program, and also the number of calls made to the underlying constraint solver, including those calls used to produce the summaries. Our evaluations are run on a laptop with Intel Core i7 CPU @ 2.20GHz $\times$ 8, with 8 GB of memory. Constraint solving is done with Z3 4.3.2.

B. Results

Table IV shows the time consumption and the number of solver calls of the complete symbolic execution on the test programs. For the time consumption, first we show its actual number in the non-summarisation mode (CSE), then the relative number in the three summarisation modes, comparing with CSE. We show the numbers of total solver calls for all modes, while also explicitly showing the calls made during summarisation in the parentheses. The threshold for R_c is set to 0.8 as explained in the experiment setup. Note that the R_c value in Table IV is the *overall* ratio of all code fragments that have been summarised and whose summaries have past the threshold, not individual ones. It serves as an indicator of how the summarisation works but not necessarily reflects the actual effectiveness of the summaries. Each time a depth limit is set and symbolic execution has to finish the search in the finite search space. Starting from depth 25, we increment the depth limit by 5 each time to see how different approaches perform as the search space grows and some of the code fragments repeatedly show up. We lower the depth requirement for 3 of the programs (`due_date`, `kmp` and `matrices`) due to the execution time being excessively long and hard for observations. A summary is shown at the end of the table. The second-last row is the average of the percentages of all tests, while the weighted average is accounting for the total number over *all* runs (expressed as a percentage). For those cases where no suitable summary is found, R_c is taken as 1 for the calculation of the average ratios.

C. Discussion

1) *General Performance*: The results from Table IV demonstrate the path explosion in the programs we evaluate, showing that they are non-trivial, albeit small. Additionally, they

show the strong correlation between the symbolic execution time and number of solver calls, as expected.

Regarding summarisation, the results demonstrate that fine-grained summary improves symbolic execution time in many cases, mostly by reducing the number of calls sent to the constraint solver. In some cases, the change is marginal, but in others, such as the `due_date` and the `tic_tac_toe`, execution time is considerably reduced. However, in some cases, summarisation increases execution time. We take a closer look at these cases now.

In a few cases, the increase is negligible (e.g., SSUM in `bubble_sort`). In others, the increase is only at small depths (e.g., LSUM in `tic_tac_toe`). Closer analysis shows that the summaries are only used a few times (e.g., summaries from LSUM are never used in `tic_tac_toe` when the depth is 25), but more time is taken to generate summaries than is saved by using them. In all cases except LSUM in `bubble_sort`, this issue disappears as depth increases, providing more opportunities for summaries to be used.

An interesting discussion point is the cases in which summarisation uses most of the solver calls. This is due to the presence of unbounded nested loops. LSUM, which does small symbolic execution on these nested loops, keeps unfolding the loops to generate more summaries, which causes this problem. We can tell from the R_c values that LSUM prefers to summarise larger code fragments, as they often have larger denominators. In such cases, we suggest the use of the more conservative LASUM mode, which performs well on these unbound loops; e.g., in the `bubble_sort` case.

There are some unusual results for `tic_tac_toe`. For LSUM/LASUM, the number of solver calls does not increase when the depth increases from 30 to 35 (similarly 40 to 45). This is because the program contains a large loop (a characteristic of a round-to-round game program), and the increase in depth does not create any new feasible paths (the ends of search of depth 30 and depth 35 are “trapped” in the same iteration where the game is unfinished). The summaries here are long enough that they pass this threshold, and without solving any constraint, our approach calculates that the game cannot finish between depth 30 and 35. It is an inspiration to us that with the summaries gathering information from a larger range of code, we could derive better heuristics for directed search.

2) *Strategy Comparison*: Through the comparison of three different summarisation strategies, we find that overall, the acyclic summarisation mode LASUM is the most effective, showing $\approx 24\%$ average time reduction of CSE, and over 36% reduction on the weighted average time. However, in several cases, the LSUM mode is certainly more efficient (e.g., `kmp`), although the LSUM mode takes more time for summarisation in these extreme cases. SSUM performs similar to standard CSE, however, it still successfully summarises small parts of the code and its costs is almost zero.

Fine-grained summarisation tends to perform better as the depth grows (even for LSUM in `bubble_sort`). As well as observing the trend in the columns, this can be seen by comparing the weighted average values, which take into account the actual time saved overall (genuinely meaning rows with larger time have larger weights). These results show that

TABLE IV. SYMBOLIC EXECUTION TIME AND SOLVER CALLS. THE NUMBERS IN PARENTHESES ARE THE SUMMARISATION OVERHEAD.

Program	Depth	Execution time				# Solver calls				R_c (< 0.8)		
		CSE (s)	LSUM	LASUM	SSUM	CSE	LSUM	LASUM	SSUM	LSUM	LASUM	SSUM
binary_search	25	2.4	92%	65%	67%	342	301 (5)	301 (5)	342 (0)	3/4	3/4	0/0
	30	6.4	86%	86%	97%	787	696 (5)	696 (5)	787 (0)	3/4	3/4	0/0
	35	19.6	89%	89%	98%	1.8 k	1.6 k (5)	1.6 k (5)	1.8 k (0)	3/4	3/4	0/0
	40	74.5	88%	72%	113%	3.9 k	3.5 k (5)	3.5 k (5)	3.9 k (0)	3/4	3/4	0/0
	45	341.9	65%	65%	99%	8.6 k	7.6 k (5)	7.6 k (5)	8.6 k (0)	3/4	3/4	0/0
bubble_sort	25	2.6	177%	71%	92%	312	327 (237)	219 (5)	312 (0)	107/237	2/3	0/0
	30	14.2	151%	72%	102%	862	877 (635)	588 (5)	862 (0)	285/635	2/3	0/0
	35	64.1	154%	64%	100%	2.3 k	2.4 k (1.7 k)	1.6 k (5)	2.3 k (0)	778/1.7 k	2/3	0/0
	40	218.2	162%	78%	107%	6.3 k	6.6 k (4.7 k)	4.2 k (5)	6.3 k (0)	2.0 k/4.7 k	2/3	0/0
	45	1.4 k	130%	60%	92%	18.3 k	17.8 k (12.7 k)	12.5 k (5)	18.4 k (0)	3.5 k/12.7 k	2/3	0/0
cba_example	25	0.9	63%	61%	91%	473	326 (7)	326 (7)	473 (0)	4/6	4/6	0/0
	30	3.0	66%	62%	90%	1.3 k	874 (7)	874 (7)	1.3 k (0)	4/6	4/6	0/0
	35	11.5	64%	65%	96%	3.5 k	2.4 k (7)	2.4 k (7)	3.5 k (0)	4/6	4/6	0/0
	40	42.1	64%	65%	99%	9.8 k	6.5 k (7)	6.5 k (7)	9.8 k (0)	4/6	4/6	0/0
	45	298.0	68%	61%	92%	35.0 k	22.9 k (7)	22.9 k (7)	35.0 k (0)	4/6	4/6	0/0
due_date	20	762.8	38%	45%	41%	218	173 (21)	173 (21)	222 (13)	12/21	12/21	8/12
	25	9.2 k	8%	10%	17%	603	402 (21)	402 (21)	555 (13)	12/21	12/21	8/12
kmp	20	2.6	125%	70%	66%	362	251 (54)	369 (7)	362 (0)	33/51	0/0	0/0
	25	14.1	92%	100%	98%	1.7 k	1.0 k (70)	1.7 k (7)	1.7 k (0)	43/67	0/0	0/0
	30	158.1	78%	101%	116%	7.8 k	4.4 k (84)	7.8 k (7)	7.8 k (0)	53/81	0/0	0/0
	35	1.1 k	82%	98%	98%	34.2 k	19.3 k (100)	34.2 k (7)	34.2 k (0)	63/97	0/0	0/0
	40	6.7 k	80%	99%	96%	143.7 k	81.7 k (114)	143.7 k (7)	143.7 k (0)	73/111	0/0	0/0
matrices	15	1.2 k	92%	98%	91%	103	110 (62)	114 (11)	103 (0)	24/59	0/0	0/0
	20	28.3 k	78%	100%	98%	372	273 (112)	383 (11)	372 (0)	45/109	0/0	0/0
tic_tac_toe	25	8.5	222%	220%	66%	592	882 (774)	882 (774)	448 (36)	198/774	198/774	16/35
	30	32.6	124%	124%	102%	2.1 k	1.1 k (774)	1.1 k (774)	1.9 k (36)	198/774	198/774	16/35
	35	72.3	59%	52%	86%	5.4 k	1.1 k (774)	1.1 k (774)	4.0 k (36)	198/774	198/774	16/35
	40	189.4	52%	60%	97%	15.0 k	5.4 k (774)	5.4 k (774)	13.7 k (36)	198/774	198/774	16/35
	45	810.6	17%	17%	82%	39.5 k	5.4 k (774)	5.4 k (774)	31.5 k (36)	198/774	198/774	16/35
wumpus	25	3.4	122%	94%	60%	3.8 k	5.1 k (4.6 k)	2.2 k (1.3 k)	2.0 k (486)	317/3.4 k	505/1.3 k	384/485
	30	11.5	118%	52%	109%	13.0 k	15.1 k (14.0 k)	5.2 k (2.3 k)	13.5 k (511)	403/10.5 k	343/2.1 k	2/3
	35	30.3	87%	54%	103%	31.3 k	28.7 k (28.2 k)	15.6 k (4.5 k)	31.8 k (511)	479/27.6 k	344/4.2 k	2/3
	40	71.5	75%	61%	102%	64.0 k	53.0 k (53.0 k)	41.2 k (9.6 k)	64.5 k (511)	618/53.0 k	345/9.2 k	2/3
	45	185.0	64%	79%	106%	113.8 k	84.5 k (84.4 k)	90.4 k (20.7 k)	114.3 k (511)	841/84.4 k	347/20.4 k	2/3
Average	-	-	92%	76%	90%	-	79% (34%)	75% (10%)	96% (1%)	0.46	0.60	0.86
Weighted average	-	-	68%	63%	86%	-	58% (54%)	64% (10%)	85% (0.5%)	0.05	0.07	0.70

LSUM and LASUM both work more effectively on larger scales, and that LSUM in particular saves more time as the depth increases.

VI. RELATED WORK

Inspired by compositional symbolic execution [1], [2], [3] and automatic partial loop summarisation [14], we have looked into summarisation of code segments. Our work differs from the partial loop summarisation technique. First, partial loop summarisation only takes effect provided with correct dynamic execution monitoring (for loop detection) and invariant guessing, while our summarisation uses sound local analysis. Hence the critical problem of partial loop summarisation is how to extract invariants from detected loops dynamically, not how to establish symbolic execution using summaries. Second, we summarise code fragments other than loops, enabling compositional symbolic execution (CSE) based on pure summaries.

Existing CSE approaches are closely related to our work. Summarisation was first used by Godefroid and Klarlund [18], implemented in SMART [1], and later extended by Anand et al. [2]. SMART generates function-level summaries on intraprocedural paths, and only computes summaries on demand. We generalise this idea by enabling incomplete, local summaries to be composed, when a certain demand is specified. SMART also achieves intelligent pruning of unrelated paths that cannot lead to the target. A more selective CSE is presented by Christakis and Godefroid [3], where only function summaries identified as *low-complexity* are recorded, which is very relevant to our target selection guideline. As we have

discussed, function-level summaries used in the mentioned work are not fully exploiting the ability of CSE in handling path explosion.

Ma et al. [4] investigate backward symbolic execution using function call chain tracing. This is a function-wise backward reasoning. Within a function, forward symbolic execution is used, but then summaries are used to compose function call sequences in a backwards manner. Thummalapenta et al. [5] discuss method sequence generation for object-oriented testing, which implements the idea of back-tracing in method call chains. Dinges and Agha [6] describe *symcretic execution*, the first attempt of combining concrete execution with backward reasoning. They use forward heuristic search when BSE cannot reason about particular parts of the code. Forward heuristic search [19], [20], [4], [17], [21] is another part of the story of *directed symbolic execution* besides BSE.

Yi et al. [22] introduce an interesting redundant path elimination approach utilising the idea of *postcondition*. More specifically, they achieve path pruning by detecting common suffixes of paths.

Symbolic execution has come a long way since the seminal work by King [23]. The current trend of the development of symbolic execution has seen several features, including: interchangeable modes [24], [13], [6], doubled direction [4], [6], and alternative state representation [25]. There is also extensive research specifically into symbolic execution of loops [14], [17], [26], [27], [28]. These approaches are mostly complementary to our work, and therefore candidate technologies for improving CSE.

VII. CONCLUSIONS

This paper introduces the idea of *fine-grained* summarisation, with which one can establish compositional symbolic execution that helps to mitigate path explosion. The fine-grained summary approach attempts to prune redundant solver calls from *any* target code fragment. We provide guidelines for code fragment selection, as well as three strategies with different qualities. The experiment shows promising results when we combine this technique with BSE, helping to improve execution time by reducing the number of calls made to the underlying constraint solver. However, the summarisation strategy used is critical to the effectiveness of the approach.

Given the positive results seen in our experiments, our next step will be to implement our ideas with an existing symbolic execution tool, such as KLEE [29], to allow us to evaluate the approach on industrial-scale examples. Our future direction will be to look into the problem of forward targeted symbolic execution, in which we aim use the fine-grained summaries in combination with heuristic search to limit the number of paths assessed. Further, to increase scalability, partial summarisation or dynamic-execution-assisted summarisation may be introduced.

REFERENCES

- [1] P. Godefroid, “Compositional dynamic test generation,” in *ACM SIGPLAN Notices*, vol. 42, no. 1. ACM, 2007, pp. 47–54.
- [2] S. Anand, P. Godefroid, and N. Tillmann, “Demand-driven compositional symbolic execution,” in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. LNCS, C. R. Ramakrishnan and J. Rehof, Eds. Springer, 2008, vol. 4963, pp. 367–381.
- [3] M. Christakis and P. Godefroid, “IC-Cut: A compositional search strategy for dynamic test generation,” Microsoft Research Redmond, Tech. Rep. MSR-TR-2015-10, February 2015, revised May 2015.
- [4] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks, “Directed symbolic execution,” in *Static Analysis*, ser. LNCS, E. Yahav, Ed. Springer, 2011, vol. 6887, pp. 95–111.
- [5] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and Z. Su, “Synthesizing method sequences for high-coverage testing,” in *Proceedings of the ACM International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*. ACM, 2011, pp. 189–206.
- [6] P. Dinges and G. Agha, “Targeted test input generation using symbolic-concrete backward execution,” in *Proceedings of the 29th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 2014, pp. 31–36.
- [7] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, “Efficiently computing static single assignment form and the control dependence graph,” *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, 1991.
- [8] P. Godefroid, N. Klarlund, and K. Sen, “DART: Directed automated random testing,” in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2005, pp. 213–223.
- [9] P. Godefroid, M. Y. Levin, and D. Molnar, “Automated whitebox fuzz testing,” in *Proceedings of the 15th Annual Network & Distributed System Security Symposium (NDSS)*, vol. 8, 2008, pp. 151–166.
- [10] N. Tillmann and J. De Halleux, “Pex—white box test generation for .NET,” in *Tests and Proofs*, ser. LNCS, B. Beckert and R. Hähnle, Eds. Springer, 2008, vol. 4966, pp. 134–153.
- [11] K. Sen and G. Agha, “CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools,” in *Computer Aided Verification*, ser. LNCS, T. Ball and R. B. Jones, Eds., vol. 4144. Springer, 2006, pp. 419–423.
- [12] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun, “jFuzz: A concolic whitebox fuzzer for Java,” in *NASA Formal Methods*, 2009, pp. 121–125.
- [13] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, “Enhancing symbolic execution with veritesting,” in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, 2014, pp. 1083–1094.
- [14] P. Godefroid and D. Luchaup, “Automatic partial loop summarization in dynamic test generation,” in *Proceedings of the 31st International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2011, pp. 23–33.
- [15] P. Boonstoppel, C. Cadar, and D. Engler, “RWset: Attacking path explosion in constraint-based test generation,” in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. LNCS, C. R. Ramakrishnan and J. Rehof, Eds. Springer, 2008, vol. 4963, pp. 351–366.
- [16] L. De Moura and N. Bjørner, “Z3: An efficient SMT solver,” in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. LNCS, C. R. Ramakrishnan and J. Rehof, Eds. Springer, 2008, vol. 4963, pp. 337–340.
- [17] J. Obdržálek and M. Trtík, “Efficient loop navigation for symbolic execution,” in *Automated Technology for Verification and Analysis*, ser. LNCS, T. Bultan and P.-A. Hsiung, Eds. Springer, 2011, vol. 6996, pp. 453–462.
- [18] P. Godefroid and N. Klarlund, “Software model checking: Searching for computations in the abstract or the concrete,” in *Integrated Formal Methods*, ser. LNCS, J. Romijn, G. Smith, and J. van de Pol, Eds., vol. 3771. Springer, 2005, pp. 20–32.
- [19] P. McMinn, “Search-based software test data generation: A survey,” *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.
- [20] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, “Fitness-guided path exploration in dynamic symbolic execution,” in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*. IEEE, 2009, pp. 359–368.
- [21] J. Strejček and M. Trtík, “Abstracting path conditions,” in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2012, pp. 155–165.
- [22] Q. Yi, Z. Yang, S. Guo, C. Wang, J. Liu, and C. Zhao, “Postconditioned symbolic execution,” in *Software Testing, Verification and Validation (ICST 2015)*. IEEE, 2015, pp. 1–10.
- [23] J. C. King, “Symbolic execution and program testing,” *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, July 1976.
- [24] V. Chipounov, V. Kuznetsov, and G. Candea, “S2E: A platform for in-vivo multi-path analysis of software systems,” *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1, pp. 265–278, 2011.
- [25] K. Sen, G. Necula, L. Gong, and P. W. Choi, “MultiSE: Multi-path symbolic execution using value summaries,” University of California at Berkeley, Tech. Rep., October 2014.
- [26] J. Jaffar, J. A. Navas, and A. E. Santosa, “Unbounded symbolic execution for program verification,” in *Runtime Verification*, ser. LNCS, S. Khurshid and K. Sen, Eds., vol. 7186. Springer, 2012, pp. 396–411.
- [27] X. Xiao, S. Li, T. Xie, and N. Tillmann, “Characteristic studies of loop problems for structural test generation via symbolic execution,” in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 246–256.
- [28] M. Trtík, “Symbolic execution and program loops,” Ph.D. dissertation, Faculty of Informatics, Masaryk University, Brno, Czech Republic, 2013.
- [29] C. Cadar, D. Dunbar, and D. Engler, “KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs,” in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, vol. 8, 2008, pp. 209–224.