MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

Compact Symbolic Execution in Slowbeast

Master's Thesis

KRISTIÁN KUMOR

Brno, Spring 2024

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Compact Symbolic Execution in Slowbeast

Master's Thesis

KRISTIÁN KUMOR

Advisor: prof. RNDr. Jan Strejček, Ph.D.

Department of Computer Science

Brno, Spring 2024


FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Kristián Kumor

Advisor: prof. RNDr. Jan Strejček, Ph.D.

Acknowledgements

I want to express my gratefulness to Jan Strejček for his time, patience, and help in getting this work done. I am also grateful for the explanations provided to me by Marek Trtík and Marek Chalupa, as well as the technical help from Martin Jonáš and Robert Konicar in the last weeks of running experiments.

Finally, I would like to acknowledge all of those who offered moral support, including friends, fellow members of Formela, and family.

Abstract

Symbolic execution is generally used for generating tests or for software verification. It works by using symbols instead of concrete program inputs and exploring all possible execution paths in a program. However, it suffers from the path explosion problem caused by path forking. This makes the classical approach suboptimal, as exploring some of the execution paths is not feasible. This work aims to mitigate the path explosion problem by summarizing the cyclic paths of loops. The result is a much smaller number of paths in the symbolic execution tree. We have implemented compact symbolic execution in Slowbeast to demonstrate the properties of this technique and tested the implementation on SV-COMP 2024 Reach-Safety benchmarks. The results confirmed our expectations of significant improvement in the amount of correctly answered benchmarks.

Keywords

Slowbeast, Symbolic execution, Compact symbolic execution, Path explosion problem, Formal verification, Program analysis, SMT-solving

Contents

1 Introduction

Symbolic execution [1] is a method for running programs on a range of inputs at once. This is useful for finding errors or verifying program correctness. Unlike testing, which only uses specific input and therefore explores only one execution path at a time, symbolic execution uses symbolic values to represent arbitrary data in a variable. Overall, it tries to explore all the program's execution paths to verify their correctness. Since there can be an infinite number of such paths in real software, symbolic execution might not finish.

Compact symbolic execution [2] is an extension of symbolic execution that attempts to partially solve one of the weak points of symbolic execution, namely the path explosion problem. This problem is the consequence of execution path forking, which can exponentially increase the number of execution paths. Developed initially by Marek Trtík [3], this method focuses on cyclic paths in the control flow graph of a program, where it uses *templates* to summarize the effects of the cyclic paths on symbolic memory and path condition. It also significantly reduces the number of execution paths that need to be executed. *Template* is a declarative parametric description, with a single parameter - κ, of all program states produced by $\kappa \geq 0$ iterations along the cyclic path, starting in the current state of symbolic execution, followed by an execution step leading outside the cyclic path [2]. However, this method is more demanding for efficient SMT-solving than classic symbolic execution, an issue described later in this work.

There are also other ways of dealing with the path explosion problem in symbolic execution. Some techniques attempt to eliminate redundant paths [4] or summarize multiple paths at once [5]. Another alternative is using compositional symbolic execution [6, 7], which uses function summaries to minimize the number of processed paths. Similarly to compact symbolic execution, there are also other efforts to solve the path explosion problem using more efficient loop processing. One of them attempts to use white-box fuzzing [8]. Meanwhile, another one is much more similar to compact symbolic execution by also using a symbolic variable for the number of loop iterations but is focused on programs in binary form [9].

The decision to incorporate this method into either JetKlee or Slowbeast happened because the original tool Bugst[1] where it was implemented by Marek Trtík, was discontinued. Slowbeast[2] is a simple interpreter and symbolic executor written in Python programming language and developed by Marek Chalupa. Its primary focus is on fast prototyping of algorithms related to symbolic execution, thus the Python language. Whereas JetKlee is a fork of Klee [10]. It is written in C++ and instead of prototyping, is more focused on performance.

Slowbeast has been chosen as the target of implementation of compact symbolic execution for two reasons. First, it would be more complicated to integrate compact symbolic execution into JetKlee. Although both tools were poorly documented, JetKlee suffers in code readability. Secondly, just like another similar tool, JetKlee[3], it is used inside Symbiotic [11], a tool developed in the Formela laboratory at the Faculty of Informatics of Masaryk University. Symbiotic is a framework that uses static analysis together with program slicing and several verification engines, one of them also being Slowbeast. Symbiotic uses Slowbeast mainly for backward symbolic execution with loop folding [12].

1. https://sourceforge.net/projects/bugst/
2. https://gitlab.com/mchalupa/slowbeast
3. https://github.com/staticafi/JetKlee

2 Preliminaries

In this chapter, we recapitulate the principles of symbolic execution. In Section 2.2, we study the overall effect of cyclic paths on memory and the so-called path condition. Section 2.3 describes the composition of symbolic states. We intentionally use a similar notation as the original paper [2] to not confuse readers who are familiar with the paper.

A *flowgraph* is a directed graph with nodes as *program locations*. Edges represent a change caused by an operation of *assignment* $(\mathrm{x} := \mathrm{z})$ or *assumption* $(\mathrm{x} < \mathrm{y})$. Additionally, edges are denoted as (a, b), where a and b are program locations. There may not be more than one edge between two states. A *path* in the flowgraph is a nonempty finite sequence of succeeding edges.

2.1 Classic symbolic execution

Unlike concrete program execution, classic symbolic execution uses *symbols* (from a set $Sym = \{\underline{a}, \underline{b}, \underline{c}, \ldots\}$) to represent all possible values of input variables of a program. A *symbolic program state* S is a triple (loc, θ, φ), where

- loc is a current location inside the program;
- θ is a *symbolic memory* containing the current values of all variables; and
- φ is a *path condition*.

More precisely, a symbolic memory is a partial function $\theta : Var \to Exp(Sym)$ mapping program variables to expressions over symbols. For example, $5 + \underline{a}$ can be such an expression. When executing an assignment instruction, the symbolic memory is updated. For example, when the instruction `x:=5+y` is executed, the resulting symbolic memory θ' will satisfy $\theta'(x) = 5 + \underline{y}$.

The path condition φ is a first-order logic formula representing a sufficient and necessary condition on symbols for the execution to arrive at the current loc using the currently executed path. The path condition initially contains *true*, and it is updated upon branching

statements, like `if` statements or loop statements. When the execution arrives at a location with an instruction that forks the execution into several paths, each containing a different condition, we instantiate these conditions using θ of the current state and check if current φ implies the validity of the conditions. If one of these implications is valid, execution continues along its branch. If none of them is valid, then for every satisfiable branch, the whole symbolic program state is forked, and its φ is updated with a conjunction of a new condition instantiated by θ. Due to these forks, symbolic execution is represented by a *symbolic execution tree*. Nodes of the tree are computed symbolic program states, and its edges represent transitions between these states.

If φ is an expression, then we write $\varphi[\kappa]$ to emphasize that it contains a parameter κ with free occurrences in φ. Similarly, if θ is a symbolic memory, then $\theta[\kappa]$ means that parameter κ appears inside symbolic memory θ.

2.2 Cyclic paths

First, we assume our flowgraphs are *reducible* [13]. This means that there is precisely one entry location a in every loop. Additionally, we assume that there are no nested loops in flowgraphs. Therefore, a is also called a *loop header*, as it is the only node with an in-edge starting from a node outside of the loop.

A cyclic path $\pi = a\gamma a$ in a flowgraph F is a path where all edges are pair-wise independent, where the a is a loop header, which is both the starting and ending location of the path.

Let N be a set of all nodes referenced in the cyclic path π and $E = \{(s_1, o_1), (s_2, o_2), ..., (s_n, o_n)\}$ be the set of all edges in F not belonging to π, but their start nodes $s_1, ..., s_n$ belong to N. We denote the nodes $\{o_1, ...o_n\}$ as *exit locations* and the edges of E as *exit edges*.

It is important to note that several cyclic paths can share the same loop header. Additionally, a path from a loop header to an exit location is called an *exit path*.

Now, it is required to formulate a *parametric path condition* $\varphi[\kappa]$ to describe the path condition after κ iterations of a cyclic path. To iterate the cycle κ times, all of the conditions (assumption operations) that

keep the execution inside the cycle must be true for every κ. Therefore, the conditions are either implied by the previous path condition or added into the path condition.

2.3 Composition of symbolic states

Let us define the composition of symbolic memories. Assume θ is a symbolic memory and φ is a formula. The value (an expression over the symbols) of a variable a inside a symbolic memory θ is denoted as $\theta(\mathtt{a})$. We write $\theta\langle\varphi\rangle$ to denote φ, where all occurrences of all symbols $\underline{a}$, are replaced by $\theta(a)$. Basically, all occurrences of $\underline{\mathtt{k}}$ inside an expression get substituted by $\theta(\mathtt{k})$, and the same happens for every other symbolic variable present in the expression.

Let θ_1 and θ_2 be two different symbolic memories. $\theta_1 \diamond \theta_2$ is a *composed symbolic memory*, that for each symbol a satisfies:

$$(\theta_1 \diamond \theta_2)(\mathtt{a}) = \theta_1\langle\theta_2(\mathtt{a})\rangle$$

Informally, the effect of θ_1 on symbolic memory is followed by the effect of θ_2.

Similarly, if S_1 and S_2 are symbolic states, we denote $S_1 \diamond S_2$ as *state composition*. Since $S_1 = (loc_1, \theta_1, \varphi_1)$ and $S_2 = (loc_2, \theta_2, \varphi_2)$, state composition satisfies:

$$S_1 \diamond S_2 = (loc_2,\ \theta_1 \diamond \theta_2,\ \varphi_1 \wedge \theta_1\langle\varphi_2\rangle)$$

Intuitively, the resulting composed symbolic state is the result of a symbolically executed code that resulted in S_1, followed by a code that resulted in S_2.

2.4 Classic symbolic execution example

In Figure 2.1 (**b**), we can see a classic symbolic execution tree of a program flowgraph from Figure 2.1 (**a**). It is a simple program that starts with a=0 and increments variable a by 4 in every iteration of the cyclic path until a is equal to some variable z or higher than some limit variable n. If a was equal to z, save the value of n into y. Otherwise, set y to zero. We assume z and n get their values as input; they can,

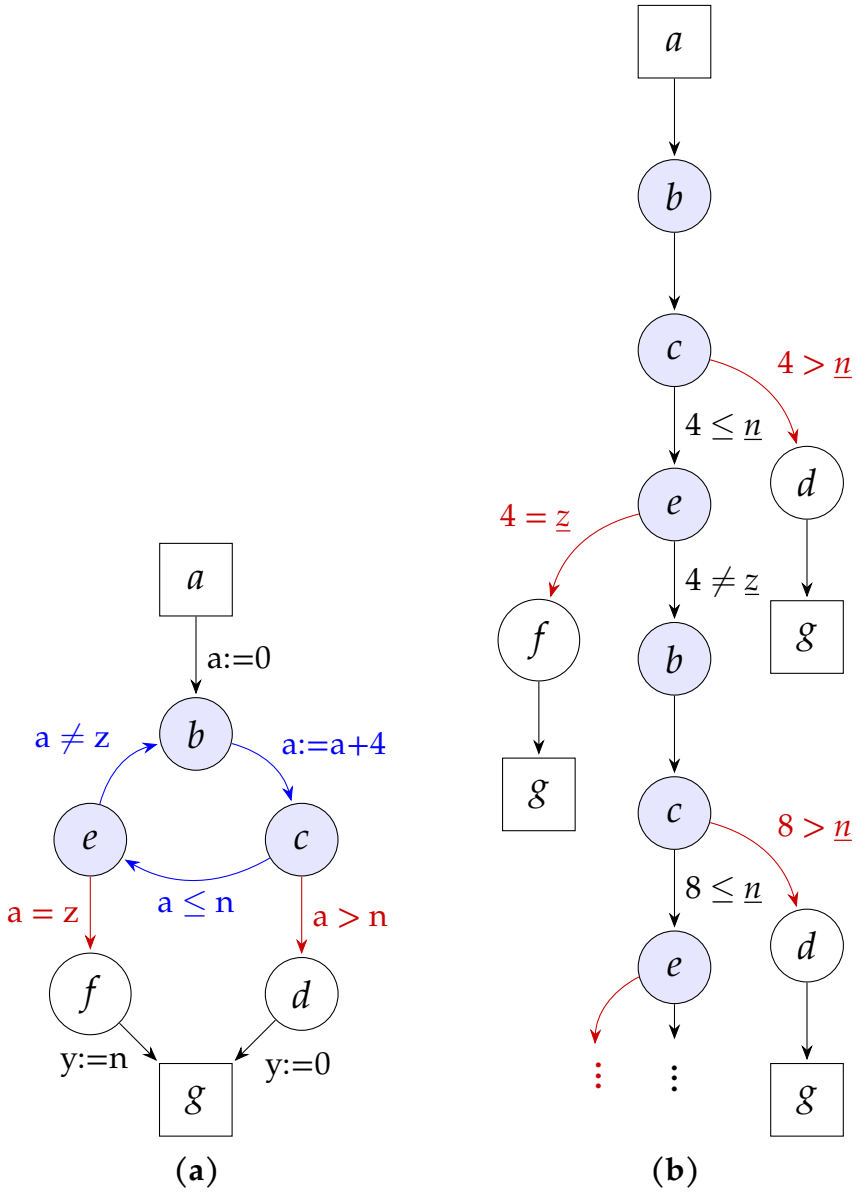

Figure 2.1: (**a**) Flowgraph of a program. A cyclic path is highlighted in blue; its exits are coloured red. (**b**) Classic symbolic execution tree of the flowgraph (**a**) with hidden assignment labels for simplicity. Edge labels represent assumption or assignment operations in every transition. Nodes with a rectangle shape can contain nodes inside them. Their main purpose is to hide unnecessary parts of the program.

therefore, have an arbitrary value. For simplicity, symbolic memory is not present. Note that in Figure 2.1, one loop can significantly expand the symbolic execution tree.

On the flowgraph of Figure 2.1 (**a**), we can see a blue-coloured loop *bce*, which has a cyclic path *bceb*, and its red-coloured exits leading to d and f. We know that the execution must iterate over the cyclic path κ times, for some $0 \leq \kappa \leq n/4$, where n is a natural number, and then leave the loop using edge *bc* and then either the first exit *cd* or the second one along *ce* and ef.

3 Compact symbolic execution

Compact symbolic execution behaves like classic symbolic execution until it detects a cyclic path. This can be done both by looking for cycle entry points before symbolic execution or during symbolic execution by checking whether the current location has already been visited.

When the symbolic execution arrives at a loop header for the first time, we compute its templates. A *template* consists of a loop header and a triple $(loc, \theta'[\kappa], \varphi'[\kappa])$ for every exit path, where the program location loc is the exit location corresponding to the exit path, parametric symbolic memory $\theta'[\kappa]$ which contains the changes of symbolic variables after κ iterations, as well as the effect of the exit path, and path condition $\varphi'[\kappa]$, which summarizes the path condition after κ iterations of the cyclic path and the sufficient and necessary condition to exit the cyclic path through the exit path. There can be a template for every cyclic path, which originates from the loop header.

An essential part of template computation is trying to define $\theta'[\kappa]$ for all variables, if possible. Of course, unchanged variables can be ignored. However, failing to define one or more variables results in template computation failure. We simply fail to summarize the effect of the cyclic path. This is done by creating a new clean state S^* and symbolically executing the cyclic path using S^* once, which results in a new state $S = (loc, \theta, \varphi)$. The original paper [2] specifies how to compute $\theta'[\kappa]$ of S for arithmetic and geometric progressions, as well as assignment:

- $\theta(a) := \underline{a} + c$ for some constant c, summarized as:
  $\theta'[\kappa](a) := \underline{a} + \kappa * c$

- $\theta(a) := \underline{a} * c$ for some constant c, summarized as:
  $\theta'[\kappa](a) := \underline{a} * c^{\kappa}$

- If $\theta(a) := e$ for some symbolic expression e, that does not refer to any symbolic variable we failed to summarize, then the following holds: $\theta'[\kappa](a) := \texttt{if } \kappa > 0 \texttt{ then } \theta[\kappa - 1]\langle e \rangle \texttt{ else } \underline{a}$

To finish the template, we also need to construct $\varphi'[\kappa]$. The general equation for the parametric path condition from the original paper [2]

is:

$$\varphi'[\kappa] = \kappa \geq 0 \ \wedge \ \forall \tau (0 \leq \tau < \kappa \implies \theta'[\tau]\langle\varphi\rangle)$$

In the next section, we present how we derived this equation from an example.

After the summarization of the cyclic path effect on symbolic variables denoted as $S' = (loc, \theta'[\kappa], \varphi'[\kappa])$, we compute the effect of all exit paths of the cyclic path. This is simply done by symbolically executing every exit path, using another clean symbolic state. Therefore, for every exit location loc', we acquire a new state $S^* = (loc', \theta^*[\kappa], \varphi^*[\kappa])$. We then check for every S^* whether its path condition $\varphi^*[\kappa]$ is satisfiable. If we cannot decide its satisfiability, we must abort the template computation. If the path condition is unsatisfiable, then this exit path is never executed, and we can ignore it. Otherwise, we finally create a template triple by state composition $S' \diamond S^*$. The final template is a pair $T = (loc, \{S' \diamond S^1, \ S' \diamond S^2, \ \dots\})$, where loc is the loop header.

After the templates are computed, we attempt to apply them to produce successor states. *Template application* happens every time symbolic execution arrives at a loop header, for which we have computed at least one template. This starts with selecting a template, but for this example, we will assume there is always only one template for every loop header. Application of a selected template begins with creating a fresh κ' parameter. Let $t = (loc, M)$ be a template, where M is a set of the aforementioned triples for every exit path and loc is its loop header. We replace all occurrences of the former κ parameter with κ' in M. If S is the current symbolic state which resulted from symbolic execution up to the loop header l, then for every triple $(loc', \theta'[\kappa'], \varphi'[\kappa']) \in M$ we compose a new state S^* that satisfies $S^* = S \diamond (loc', \theta'[\kappa'], \varphi'[\kappa'])$. Note that these triples in M satisfy the definition for symbolic states. However, they are not supposed to be symbolically executed on their own, as their origin does not make them valid for the aims of symbolic execution. They simply lack context, which is the missing symbolic execution from a program starting point to the loop header. On the contrary, these newly composed states are used as any other symbolic state in symbolic execution, as they represent the current state S extended by κ iterations of a cyclic path, followed by exiting it.

In the case of template computation failing to produce any templates, which may happen because of an unsupported operation or a memory change that cannot be summarized, we abort the template application and continue using classic symbolic execution.

3.1 Compact symbolic execution example

To find the overall effect of cyclic paths in flowgraphs, we take note of all changes in the variables on the cyclic path. In the case of Figure 2.1 (**a**), it is the repeated addition of 4 to variable a, which happens κ times and then once more when leaving the cycle. Using this, we can easily derive the symbolic value of a. We also need a valid path condition to capture this path's effect adequately.

Firstly, we must introduce symbols representing the values of variables used (read or modified) inside the cycle. These are $\underline{a}, \underline{n}, \underline{z}$. The symbols correspond to their respective values at the time of cycle entry, not the start of the program, as they can be different every time symbolic execution arrives at the node b of the flowgraph using different paths.

We will describe the effect of the cyclic path in Figure 2.1 after κ iterations:

$$\begin{aligned} \theta(a) &= \underline{a} + 4 * \kappa \\ \theta(n) &= \underline{n} \\ \theta(z) &= \underline{z} \end{aligned}$$

Now, we produce the parametric path condition $\varphi[\kappa]$. This is relatively simple, too. To iterate the cycle κ times, all of the conditions that keep the execution inside the cycle must be true for every κ. These are $a \neq z$ and $a \leq n$. Formally, the path condition will look like this:

$$\begin{gathered} \underline{a} + 4 \leq \underline{n} \ \wedge \ \underline{a} + 4 \neq \underline{z} \ \wedge \\ \wedge \ \underline{a} + 4 * 2 \leq \underline{n} \ \wedge \ \underline{a} + 4 * 2 \neq \underline{z} \ \wedge \\ \ldots \\ \wedge \ \underline{a} + 4 * \kappa \leq \underline{n} \ \wedge \ \underline{a} + 4 * \kappa \neq \underline{z} \end{gathered}$$

We can simplify this by using a universal quantifier and a fresh variable τ:

$$\forall\tau.(0 \leq \tau < \kappa \implies (\underline{a} + 4 * \tau <= \underline{n} \wedge \underline{a} + 4 * \tau \neq \underline{z}))$$

It is obvious that if κ represents the number of cyclic path iterations, it cannot be a negative number. Therefore, the final parametric path condition is:

$$\varphi[\kappa] = \kappa \geq 0 \wedge \forall\tau.(0 \leq \tau < \kappa \implies (\underline{a} + 4 * \tau <= \underline{n} \wedge \underline{a} + 4 * \tau \neq \underline{z}))$$

Now it is clear that the equation above corresponds to the general equation for parametric path condition:

$$\varphi'[\kappa] = \kappa \geq 0 \wedge \forall\tau(0 \leq \tau < \kappa \implies \theta'[\tau]\langle\varphi\rangle)$$

Note that all occurrences of κ are replaced by τ inside the path condition φ.

Next, we create a triple for every exit. In our case, the exits on the flowgraph in Figure 2.1 (**a**) are highlighted in red. To reach the exits, we get two different exit paths, *bcd* and *bcef*. Since these paths do not modify any variables except a, we get $\theta_{bcd}[\kappa] = \theta_{bcef}[\kappa] = \theta[\kappa + 1]$, where θ_{bcd} and θ_{bcef} correspond to the symbolic memory of their respective exit paths. The path condition is created as a conjunction of the resulting $\varphi[\kappa]$ we acquired above and the conditions along the exit paths:

$$\varphi_{bcd}[\kappa] = \varphi[\kappa] \wedge \underline{a} + 4 * (\kappa + 1) > \underline{n}$$
$$\varphi_{bcef}[\kappa] = \varphi[\kappa] \wedge \underline{a} + 4 * (\kappa + 1) \leq \underline{n} \wedge \underline{a} + 4 * (\kappa + 1) = \underline{z}$$

For our example in Figure 2.1, the template is defined by the loop header b and a triple for each exiting path $(d, \theta_{bcd}[\kappa], \varphi_{bcd}[\kappa])$ and $(f, \theta_{bcef}[\kappa], \varphi_{bcef}[\kappa])$. *bcef* and *bcd* are the exit paths from the loop header to the exit edges outside of the cyclic path.

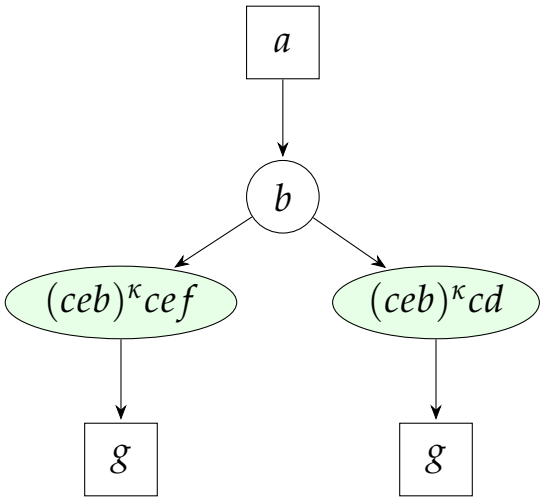

Figure 3.1: Compact symbolic execution tree derived from the flowgraph in 2.1 (**a**). Labels of the nodes coloured green represent all paths in the flowgraph that get replaced by the template application.

The triple $(d, \theta_{bcd}[\kappa'], \varphi_{bcd}[\kappa'])$ will generate a successor symbolic state $S^* = (d, \theta^*_{bcd}[\kappa'], \varphi^*_{bcd}[\kappa'])$. $\theta^*_{bcd}[\kappa']$ and $\varphi^*_{bcd}[\kappa']$ are created by composition with the current symbolic state $S = (b, \theta', \varphi')$. S is the state produced by symbolic execution up to the loop header. In our simple program from Figure 2.1, θ' is affected only by the assignment `a:=0`. This effectively means that $\underline{a}$ is replaced by 0 inside $\theta^*_{bcd}[\kappa]$ and $\varphi^*_{bcd}[\kappa]$. The symbolic execution will continue from locations d and f with the new successor states using classic symbolic execution.

This significantly reduces the size of the symbolic execution tree. An example can be seen when comparing trees in Figure 2.1 (**b**) and Figure 3.1.

3.2 Multiple templates

Now, we will emphasize how the behaviour of compact symbolic execution differs when there is more than one template present at the same loop header. Consider the example in Figure 3.2. This loop would generate two templates, one that summarizes the *bcdfeb* path, the other one the *bceb* path. We know that with sufficiently high `y`,

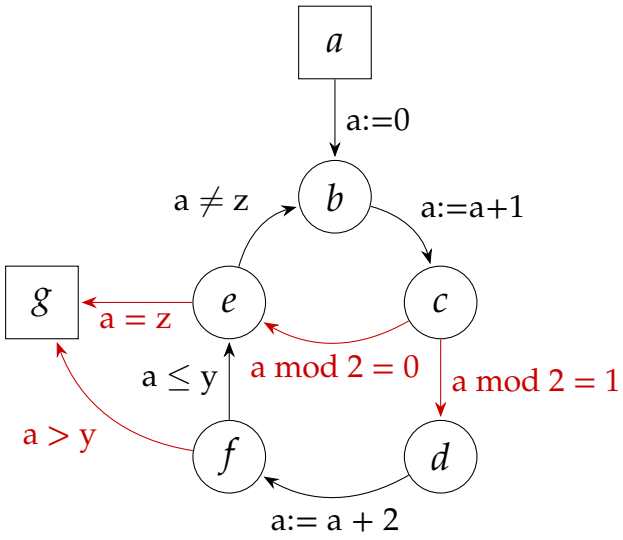

Figure 3.2: A loop with two alternative paths. The exits of the cyclic paths are coloured red. y and z are input variables. The path *bcdfeb* is taken every time a is odd, and *bceb* is taken when a is even. Note that the check of a being odd or even is taken *after* the increment by one is executed. Computation leaves the loop when a is equal to z or is greater than y.

both paths will be taken, and therefore, symbolic execution must deal with it.

Assume we want to compute a template for the cyclic path *bceb*. The first obvious exit is the edge *eg*, coloured red. Most importantly, *cd* is also an exit edge of this cyclic path. Similarly, when computing a template for the cyclic path *bcdfeb*, its exits are both edges *eg* and *fg*, as well as *ce*.

Now, we need to decide which template to apply. Searching for an optimal selection strategy is outside of the scope of this work. Therefore, we will choose randomly. Of course, selecting the same template successively twice would not be useful, as the template already summarizes κ iterations, where κ ranges from zero to (in theory) infinite. Because of that, a useful template selection strategy always chooses a template that is different from the previously selected one.

Additionally, it is noticeable that using compact symbolic execution in Figure 3.2 would have a worse result than using classic symbolic

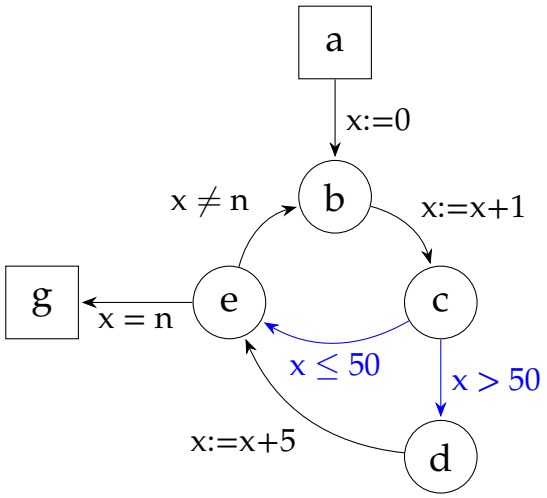

Figure 3.3: Flowgraph of a program incrementing variable `x` until it is higher or equal to `n`. If `x` is higher than 50, increment is 5 instead of 1.

execution because of both the additional overhead of repeatedly applying a template and the fact that each path is successively taken only once.

On the other hand, if there is only a single alternation between the cyclic paths, compact symbolic execution performs much better than in Figure 3.2. As an example, take a look at Figure 3.3. There is a flowgraph with a loop which increments some variable x from zero to some natural number that is higher or equal to n with a single `if` statement with condition $x > 50$. It is evident that after the cyclic path *bceb* is executed several times, it will never be executed again when variable x reaches a value higher than 50. After that, the cyclic path *bcdeb* iterates without being interrupted by the *bceb* path. Therefore, compact symbolic execution can leave this loop just after two template applications, assuming optimal template selection.

3.2.1 Loop analysis

For the purposes of template computation, we distinguish several notable loop attributes:

- **Cyclic paths** - the number of different cyclic paths originating from the same loop header that are present in the control flow

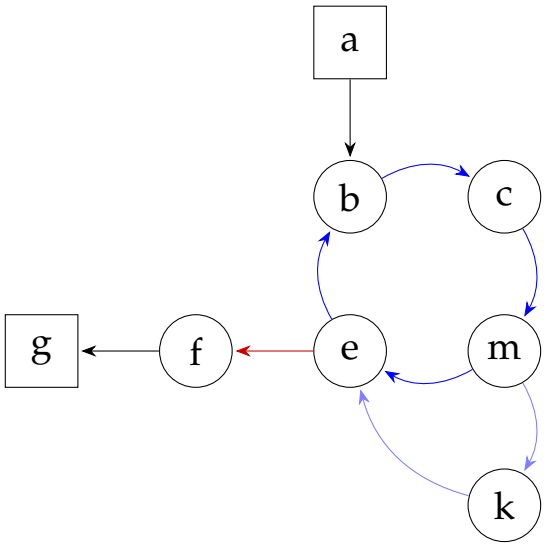

Figure 3.4: A blue-coloured loop which contains two distinct paths, with one red exit.

graph. For example, in Figure 3.4, there are two cyclic paths originating from *b* and are highlighted in blue: *bcmeb* and *bcmkeb*.

- **Exits from loop and cyclic paths** - every exit from a cyclic path has a path condition that must be satisfied by the execution to exit the path. If there are none, we have found an infinite cyclic path where there is no point in continuing symbolic execution. Still, we must check for errors, like division by zero, inside the cyclic path. Therefore, if there is a single infinite cyclic path originating from the loop header and no memory or division errors are possible when executing the cyclic path, we do not have to create any successor symbolic states. This can cut a potentially infinite symbolic execution subtree. As an example, it includes the case of a `while(true)` loop, which does not change any variables.

If there is only one cyclic path originating from a loop header, we consider the loop *simple*. Computing a template for loops that are not simple brings several challenges. Dealing with multiple cyclic paths may result in multiple templates at one location. The original paper of Slabý, Strejček, and Trtík [2] describes a solution to this issue that is good enough for cases where the different paths rarely alternate.

There is a possibility of a loop having more cyclic paths than what we can compute in reasonable time by simply having enough `if` statements inside. Computing templates for all of them could be very expensive, and most of them are likely to be invalid[1] usually because such a path will never be taken by execution. This is the reason why we decided to run classic symbolic execution for such loops instead.

1. By invalid, we mean the template computation failed.

4 Implementation

In this chapter, we first describe Slowbeast, the tool where we chose to implement compact symbolic execution. In the next section, we present our implementation in more detail.

4.1 Slowbeast

Slowbeast[1], being a symbolic executor written in Python, is focused on fast symbolic execution prototyping. It has library dependencies, such as `z3-solver` [14], `numpy`[2], and `llvmlite`. Note that a modified fork of the `llvmlite` library, made by Marek Chalupa, is required[3]. With the official library, Slowbeast would not work correctly.

Aside from classic symbolic execution, Slowbeast also contains other variants of symbolic execution, like *backward symbolic execution* (*BSE*), *k-induction* (*KindSE*), *backward symbolic execution with loop folding* (*BSELF*), or *stateful symbolic execution*. Despite the lack of documentation and rarely present type annotations, the source code of Slowbeast is fairly well-readable.

A notable technical detail related to this work is the absence of nested loop recognition in Slowbeast. Detection itself works. However, the loop structure is not analyzed, and almost no information about the loop is provided. An issue that was found in Slowbeast was the inability to detect loops with more than one arriving path. This is generally achieved by using `goto` statements outside the loop, with labels inside the loop. Since something like this rarely happens in real-world programs, it is not very important for this thesis.

The rest of this section contains a more detailed description of this tool, as it may be important for someone doing future work on this tool or extending the technique described in this work.

The input for Slowbeast is a C or LLVM [15] program file. If it is C, it is immediately compiled into LLVM. An essential part of compact symbolic execution is preprocessing, where loops are detected

1. https://gitlab.com/mchalupa/slowbeast
2. https://numpy.org/
3. https://github.com/mchalupa/llvmlite

by looking for strongly connected components and processing them further.

With regards to things of importance for compact symbolic execution, the general structure of the symbolic execution part of this tool contains:

- **Z3-solver expressions handling** - creates an abstract `Expr` object on the Z3 expressions for higher-level processing and formulae simplification. This abstracting class does not contain all the features of Z3. This became an issue since exponentiation and for-all quantifiers are required by compact symbolic execution. Therefore, effort has been put into this part of Slowbeast, too.

  The `ExpressionManager` class acts as an interface between Z3 expressions and symbolic execution classes. It is the starting point for any Z3 feature additions, like quantifiers, exponentiation or expression simplification.

  Values inside expressions may be concrete and symbolic. Concrete values are known. Usually, it is an exact number or boolean. Meanwhile, symbolic values are symbols of arbitrary value until bounded or constrained. As of the time of writing this work, Slowbeast supports bytes, boolean, bit-vector and floating-point values. All may be symbolic or concrete.

- **Base symbolic execution classes** - these classes contain very basic symbolic executors, interpreters, states and memory. Any further extensions, like forward or backward symbolic execution, are handled by child classes inside their respective folders. The class that handles all other classes and starts the whole execution is the symbolic interpreter.

  Symbolic memory in Slowbeast is quite complicated. It not only contains the global and local memory objects but also the call stack. This complicates the symbolic memory manipulation in compact symbolic execution significantly as we must modify not only the memory objects and path condition but also the call stack.

  It is also important to note that for the reduction of memory copying, all memory instances do not immediately copy all of

their contents upon calling their `copy` function. Many of their contents have `read-only` flags that do not allow simple modification. How exactly this mechanic works is not described in this work.

Regarding memory management, Slowbeast contains a `Memory` class. It contains the call stack and, most importantly, *memory objects* (`MemoryObject` class), which represent the stored values of individual symbolic variables. They are also responsible for the creation and manipulation of these memory objects.

4.2 CSE implementation

This section is focused on describing the compact symbolic execution implementation in Slowbeast, especially how the tool was modified and improved.

The `Memory` and `MemoryObject` classes inside Slowbeast are perhaps the most important ones for this work. The `Memory` class represents the entirety of the symbolic memory of a state and stores the instances of the `MemoryObject` class, which encapsulates information about individual symbolic variables and, most importantly, their current symbolic values. As we will show later, there is a need to read and manipulate the memory objects to summarize the effects of a cyclic path. When we mention memory objects, we are referring to the instances of `MemoryObject`.

As mentioned before, classic symbolic execution is used until the loop header is reached. For loop header detection, we use the functionality already implemented inside Slowbeast. Then, everything depends on whether we already have computed templates at this location and how many there are.

- **No templates** - we compute templates for this location as described in the following subsection.
- **No templates, but we already tried to compute them and failed** - in this case, we simply continue using classic symbolic execution. Failed template computation might be because of an unsupported operation or other reasons described later in this chapter.

- **Templates present** - if we have only one template, we just apply it. In the case of more templates, we choose one of them based on some strategy.

Template application yields new symbolic states, which can continue being executed using classic symbolic execution until another loop header is reached.

In the rest of this section, we are going to describe the details of template computation and application, which are the main components of compact symbolic execution.

4.2.1 Changes to memory

When computing a template, we must note changes made to symbolic variables. This is done by creating a state with a new parametric symbolic memory, with expression simplifications turned off. By parametric symbolic memory, we denote a copy of the current symbolic state, where the values of its memory objects are not copied. Instead, a new symbol representing the old value is inserted into the objects, with the intention to substitute it later and distinguish the effects of a cyclic path on the objects. This state is used by the symbolic executor to run the whole cyclic path once. The resulting state is then used for template creation, and its symbolic memory is inspected for changes to its variables.

Slowbeast works with signed and unsigned integers as well as arrays and also has some support for floating-point numbers. As described in the original paper [2], arrays of symbolic size are very tricky to implement and design and, therefore, are better left for possible future work. Floats are also not supported by our compact symbolic execution implementation due to some bugs having been detected in the original Slowbeast when running `SV-COMP` benchmarks.

It is important to note that Slowbeast uses bit-vectors from the z3-solver library for symbolic and concrete computations. Concrete computations are regular computations with concrete numbers without any unknown or symbolic values.

Our implementation, therefore, handles only integers and also not all of the mathematical operations inside cyclic paths that it tries to summarize. The original paper [2] solves addition and multiplication. While addition and subtraction are very trivial, multiplication

presented a challenge of symbolic bit-vector exponentiation. In short, the z3-solver library does not implement exponentiation on symbolic bit-vectors needed for cyclic path effect summarization of variable multiplication by a constant. This was described earlier as $\theta(x) = \underline{x} * c^{\kappa}$, where x is a symbolic variable and c is a concrete number. The exponentiation has been implemented into Slowbeast by producing formulae that check individual bits of a bit vector.

There is, of course, the question of division. Since we are working with integers that may lose information because of remainders, a repeated application of integer division does not equal what would be considered the obvious step:

$$(...((x/c)/c).../c) \neq x/c^{\kappa}$$

Therefore, if we detect division or any other operation that is not supported by compact symbolic execution, template computation is aborted, and classic symbolic execution continues.

Memory objects that are changed may also refer to values of other memory objects, constant or not. Many of these changes require special treatment. For better readability, we include a list:

- **Concrete assignment** - assigns a concrete value v to the memory object. We simply note this change as an assignment of v into the memory object.

- **Symbolic assignment** - assigns a symbolic value s to the memory object. We also note this change as an assignment of s into the memory object since it does not refer to a symbolic variable and is usually just a representation of an arbitrary input value, which is just a constant in non-symbolic execution.

- **Binary/unary operation with a concrete number** - usually a repeated addition/subtraction/multiplication by the same concrete number κ-times. This is where the operation summarization equations are used.

- **Operations referring to a constant memory object** - by constant, we mean an object that is constant on the cyclic path, not

necessarily outside of it. Here, we can simply substitute the symbolic/concrete value contained by the referred memory object at the time of loop header entry.

- **Operations referring to a changing memory object** - in this case, the object is changed in every iteration. It is important to note when and how does this object change. If we cannot derive the summarized value of the referred object first, then the template computation aborts.

Other changes are not currently supported and result in template computation failure.

4.2.2 Template creation

First, we check the conditions needed to escape the loop. We know that to have any possibility of leaving a cyclic path of this loop, at least one variable referenced inside the conditions must be changing. If there is no such variable in any cyclic path, then the loop is very likely infinite unless it contains division by zero or other runtime errors.

Most of the implemented algorithm for template creation is identical to the theory presented above. The implementation differs from the theory in that it does not create triples for every exit path. Instead, we only summarize the κ iterations along the cyclic path and leave the exit paths for individual template applications. This is still valid, as the result of this approach is the same, i.e., the resulting state contains both the effect of the κ iterations of the cyclic path and the effect of the exit path. Even the satisfiability of the successor states is automatically checked by the executor, which detects errors on the path if their path condition is not satisfiable. The main difference and advantage of this approach is that, when applying templates, instead of a state composition for every exit path, we only have one. Additionally, we only need to remember the exit paths and one triple, which represents the κ iterations of the cyclic path, instead of a triple for every exit path. This way, we also make fewer direct changes to the states, which lowers the chance of bugs.

As mentioned before, the template computation may fail due to unsupported memory operations or the SMT solver failing when executing the cyclic path. In such a case, our implementation attempts

Algorithm 1 computeTemplate

Input: a cycle $(\pi,\ header,\ exitPaths)$ and a clean state s
Output: a template $(exitPaths, s')$ or `None`

1: $(loc, \theta, \varphi) \leftarrow$ `executePath`(s, π)
2: $\theta_\pi[\kappa] \leftarrow$ `summarizeChanges`(θ)
3: **if** $\theta_\pi[\kappa]$ is `None` **then**
4: **return** `None`
5: **end if**
6: $\varphi_\pi[\kappa] \leftarrow \kappa \geq 0 \ \wedge\ \forall\tau(0 \leq \tau < \kappa \implies \theta_\pi[\tau]\langle\varphi\rangle)$
7: **return** $(exitPaths,\ (header,\ \theta_\pi[\kappa],\ \varphi_\pi[\kappa]))$

to compute other templates originating from this loop header. If no templates are produced, the loop header location is marked as already computed, and no template application will happen on this location.

The function `executePath` in Algorithm 1 uses symbolic executor to execute a given path in the analyzed program by using a provided symbolic state as the initial state. Additionally, `summarizeChanges` returns a parametric symbolic memory with summarized changes as described in Section 4.2.1, or `None` if summarization fails.

4.2.3 Template application

If template computation is successful, we can choose one of them. Otherwise, we continue using classic symbolic execution. With a chosen template, we compose the changes stored in the template with our current symbolic state and replace every occurrence of the old κ present in the template with a fresh κ' inside the newly composed symbolic state s'. The fresh κ' parameter is created in Algorithm 2 by the `getFreshKappa` function. This new κ is required to avoid collisions with other κ parameters since two executions of the same cyclic path may differ in the number of iterations. Given a symbolic expression, the function `sat` returns whether the expression is satisfiable, unsatisfiable, or unknown.

Now, using the composed symbolic state s', we generate a new exit state for each exit path in the cyclic path. This is done by making a copy of our composed symbolic state and letting the executor execute

Algorithm 2 applyTemplate

Input: a chosen template $(exitPaths, (loc, \theta_\pi[\kappa], \varphi_\pi[\kappa]))$ and a current symbolic state s
Output: a set of new successor symbolic states S

1: Let $s_\pi = (loc, \theta_\pi[\kappa], \varphi_\pi[\kappa])$
2: $s' \leftarrow s \diamond s_\pi$
3: $\kappa' \leftarrow$ `getFreshKappa()`
4: Substitute all occurrences of the former κ parameter in s' by κ'
5: $S \leftarrow \{\}$
6: **foreach** $exitPath \in exitPaths$ **do**
7: $(exitLoc, \theta[\kappa'], \varphi[\kappa']) \leftarrow$ `executePath`$(s', exitPath)$
8: **if** `sat`$(\varphi[\kappa']) =$ `SAT` **then**
9: $S \leftarrow S \cup \{(exitLoc, \theta[\kappa'], \varphi[\kappa'])\}$
10: **end if**
11: **if** `sat`$(\varphi[\kappa']) =$ `UNKNOWN` **then**
12: **return** $\{\}$
13: **end if**
14: **end foreach**
15: **return** S

the exit path from this state. If this exit from the cyclic path is viable, i.e., the φ of the returned state is satisfiable, then this state is added to the queue of symbolic states ready for execution. Specifically inside Slowbeast, a path condition φ is satisfiable when the executor does not return a state which terminated early or resulted in an error. We omitted this technicality in Algorithm 2 for simplicity.

If there are no successfully generated states, running an exit path produces an exception, or the SMT solver exceeds its time limit while checking exit path satisfiability[4], then no new states are returned, and the loop is executed via classic symbolic execution.

4.2.4 SMT solver

Because of the added $\forall$ quantifier in expressions, the time the SMT solver needs to return an answer increases significantly. For cyclic

4. This results in an `UNKNOWN` answer.

paths with a small concrete number of iterations, it may be faster not to use templates at all. Unfortunately, the SMT solver choice or symbolic expression optimizations are not part of this work. Therefore, we limit ourselves with an added time limit on the SMT solver whenever we use the symbolic executor to check the satisfiability of a path condition or to execute paths. When the limit is exceeded, we abort the template computation or application.

Since compact symbolic execution relies heavily on the SMT solver, choosing a different solver than the used `z3-solver` library may significantly impact the performance and even results in comparison to classic symbolic execution.

4.2.5 Installation and running

Before installing, the requirements are Python 3.10+ interpreter with several libraries that need to be installed: Marek Chalupa's fork of `llvmlite`[5], latest `z3-solver` and `numpy`. After that, clone `benchexec`[6], which we used for benchmarking, although there may be other alternatives to this framework, finally, clone our fork of Slowbeast[7].

The tool can be run with the `sb` executable. For example, to run Slowbeast with backward symbolic execution on Ubuntu, use the following command:

`./sb -bse` ⟨`path to C/LLVM file`⟩

The executable itself provides the complete list of arguments. Regarding our implementation, these are the newly added command line arguments:

- **`cse`**: Turns on compact symbolic execution.
- **`cse-multiple-temp`**: Allows compact symbolic execution to process multiple cyclic paths at one loop header. Without this argument, Slowbeast uses classic symbolic execution when it detects a loop header with multiple cyclic paths. Therefore, only simple loops are summarized. This may be useful in cases where running compact symbolic execution would have worse results.

5. https://github.com/mchalupa/llvmlite
6. https://github.com/sosy-lab/benchexec
7. https://gitlab.fi.muni.cz/xkumor/slowbeastcse

5 Experiments

The implementation of this work is tested on the reach safety category of SV-COMP 2024 benchmarks[1]. The following subcategories were included:

- BitVectors
- Floats
- Heap
- Loops
- Recursive
- Sequentialized
- XCSP
- Arrays
- ControlFlow
- Hardware
- ProductLines

Benchmark subcategories that were not included:

- ECA - Slowbeast returns no results for any of these benchmarks. Most of them were timeouts.
- Combinations - Recent addition to SV-COMP. None of these benchmarks were decided by Slowbeast, and all of them timed out. Therefore, there is no reason to include them in the analysis.
- Hardness - Also a recent addition to the competition. Slowbeast was not accommodated to deal with these benchmarks.

1. These were downloaded from https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/tree/svcomp24.

- Fuzzle - Only one of these benchmarks was decided by Slowbeast, both with and without the compact symbolic execution. It is, therefore, not interesting.

We ran the experiments with clang-13 as the compiler of C files in Slowbeast. While running experiments, we noticed that the results differ between different LLVM versions. However, this is to be expected from a tool that analyzes LLVM IR. All benchmarks were executed with 1 CPU core and 12 GB of RAM memory on an AMD EPYC 7371 16-core processor with a 4-minute timeout for every benchmark.

5.1 Running benchmarks

The requirement to run `benchexec` with `SV-COMP` benchmarks is to download the benchmarks themselves from the `SV-COMP` website[2]. There is a `benchexec` folder inside our fork of Slowbeast containing two tool-info modules and an `example.xml` file, where the user chooses which benchmarks to run, with additional parameters like timeout, memory limit, and number of CPU cores and which tool-info module to use. The provided `slowbeast.py` tool-info module should be sufficient for local benchmark running. It needs to be placed inside the `/benchexec/benchexec/tools` directory to work.

5.2 Results

We refer to **Slowbeast with CSE** as our version of Slowbeast with compact symbolic execution. Meanwhile, **Slowbeast** is the unmodified version of Slowbeast from Marek Chalupa[3]. The aforementioned tool-info module parses the output of the tool in question and returns either `false(unreach-call)`, `true`, `unknown`, `TIMEOUT`, or `ERROR(...)` depending on the parsed result, tool exit signal, and the time of computation.

The tables presented here contain the number of correctly decided benchmarks, separately for both true and false, and the incorrect answer count (a tool returned `true` for a benchmark which expects `false`

2. https://sv-comp.sosy-lab.org/
3. https://gitlab.com/mchalupa/slowbeast

All (5324 total)	Slowbeast	Slowbeast CSE
total correct	1011	1066
correct true	720	786
correct false	291	280
incorrect answer	7	11
total time (s)	∼745000	∼709000
total memory (GB)	∼827	∼5520

Table 5.1: Summary of the benchmark results of all included subcategories containing 5324 benchmarks in total.

or vice versa). Additionally, we include the total time used in seconds and memory in gigabytes.

From Table 5.1 it is evident that our implementation is significantly more demanding for memory. However, there is a significant time reduction. Additionally, **Slowbeast CSE** has 55 more correctly decided benchmarks. Note that this does not mean that compact symbolic execution helped to decide 55 benchmarks that classic symbolic execution could not as such number is higher. In total, **Slowbeast CSE** correctly decided 106 benchmarks, where **Slowbeast** timed out and **Slowbeast** correctly answered 51 benchmarks, where **Slowbeast CSE** failed or timed out.

Furthermore, there is a higher number of correct `false` answers from classic symbolic execution. This comes as no surprise since it is easier to find errors by executing individual iterations rather than attempting to summarize the whole loop. However, compact symbolic execution is more successful at exploring the whole program, which yields more `true` answers.

In Figure 5.1, we can see a clearer difference between classic and compact symbolic execution. Note the performance overhead caused by compact symbolic execution for simple benchmarks decidable in under one second and the already better overall results after four seconds.

5.2.1 Incorrect answers

Out of the 11 incorrect answers in total, 7 of them are shared by both tools. Out of the other four answers, (`copysign` and `remainder`) are

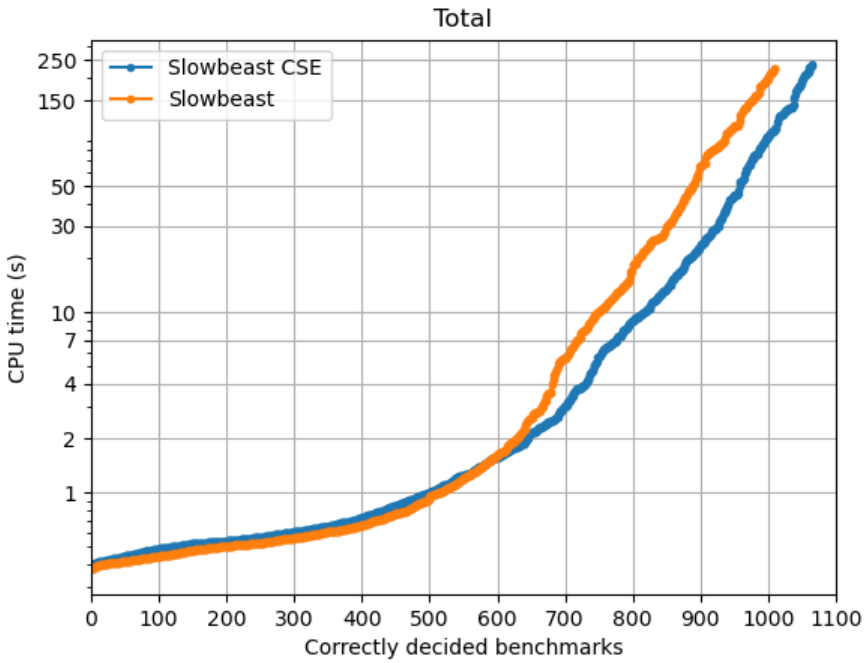

Figure 5.1: Log-scaled plot of all correctly decided benchmarks in time. Over 700 benchmarks are decided in under 4 seconds of CPU time.

from the Floats subcategory and seem to be caused by the original Slowbeast. While **Slowbeast CSE** incorrectly returns `false`, **Slowbeast** returns `ERROR(returned 1, ERROR)`. Further analysis of the output logs confirmed that both tools incorrectly find errors in the benchmarks, but the original Slowbeast also crashes on an exception. Also, neither of these benchmarks contains a loop header.

The two remaining incorrect answers are `lcm2_unwindbound10` and `lcm2_unwindbound5`. **Slowbeast** correctly answers `true` for the latter benchmark and returns `TIMEOUT` on the former benchmark. We failed to find the cause of this result returned by **Slowbeast CSE**.

5.2.2 Subcategory differences

Some of the subcategories had no differences in their results:

- Arrays

- Heap
- ControlFlow
- ProductLines
- XCSP
- Recursive
- Floats - if we ignore the benchmarks that crash in **Slowbeast**, out of which two were answered incorrectly, while **Slowbeast CSE** returns `unknown` for the rest, then there are no other differences in this subcategory.

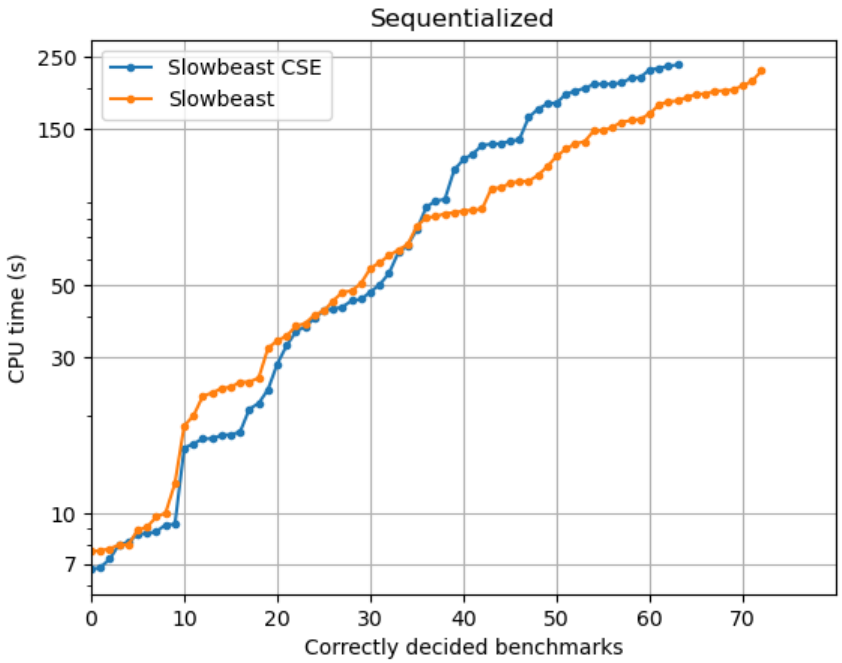

Figure 5.2: Log-scaled plot of correctly decided benchmarks for the Sequentialized subcategory.

Loops (773 total)	Slowbeast	Slowbeast with CSE
total correct	380	452
correct true	272	338
correct false	108	114
incorrect answer	1	3
total time (s)	∼90800	∼70700
total memory (GB)	∼85	∼105

Table 5.2: Summary of the benchmark results of the Loops category only, which contains 773 benchmarks in total.

The rest of the subcategories have small but insignificant differences, with less than ten correct results in comparison, with the exception of the Sequentialized and Loops subcategories.

The Hardware subcategory also had significant differences, but not in the number of correctly decided benchmarks. There are 141 benchmarks where **Slowbeast CSE** returned `ERROR`. The error was caused by an exceeded recursion depth in Python because of our implementation of nested loop recognition into our fork of Slowbeast. On the contrary, **Slowbeast** timed out on all of these benchmarks. Since there are clearly nested loops, our implementation of compact symbolic execution would not improve the results.

The Sequentialized subcategory has 584 benchmarks in total, and only 73 of them were decided by **Slowbeast**, 64 by **Slowbeast CSE**. This subcategory contains sequentialized concurrent programs, which are derived from SystemC programs, as described on the `SV-COMP` website[4]. It is clear that these benchmarks are much more demanding than the average. No templates were successfully applied in this subcategory. Therefore, Figure 5.2 shows primarily the overhead required to attempt template computation, which fails. The occasional better results of **Slowbeast CSE** are caused by time inaccuracies in benchmarking, most likely by Python hashing.

Our expectation was that the main difference would be in the Loops subcategory since the benchmarks focus on loops. This has been confirmed by the results in Table 5.2 and Figure 5.3.

4. https://sv-comp.sosy-lab.org/2024/benchmarks.php

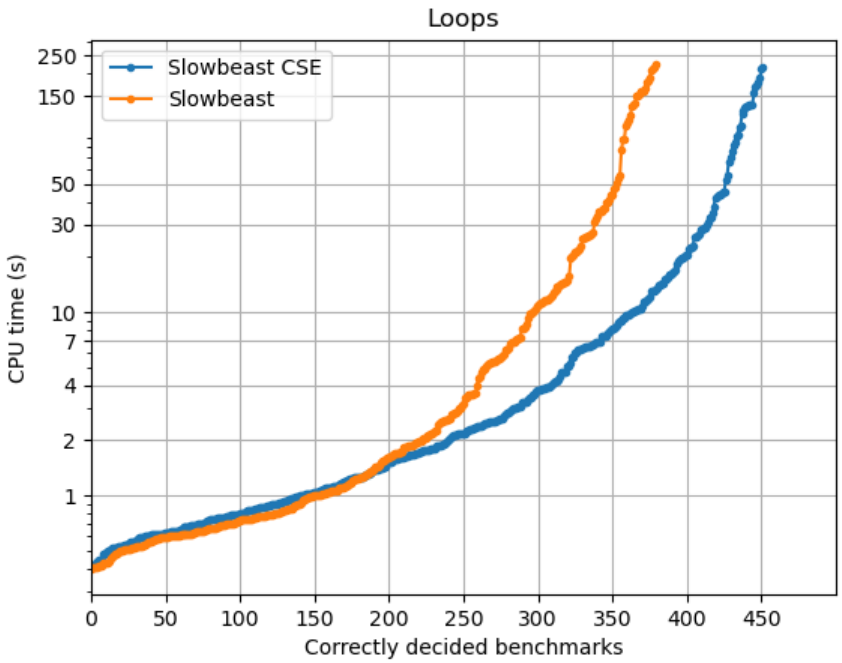

Figure 5.3: Log-scaled plot of correctly decided benchmarks for the Loops subcategory.

Figure 5.3 shows that symbolic execution performs better after just two seconds, and the difference continually increases with time.

We include Figure 5.4 to show the difference exclusively on benchmarks where compact symbolic execution created at least one template. All dots with a coordinate near 240 are timeouts. Based on the character of the selected benchmarks, it is more common for the compact symbolic execution to be slower than classic symbolic execution when both methods terminate in time with only a few exceptions in the first second of computation. However, it is noticeable that there is a large number of benchmarks decided by **Slowbeast CSE** in the first two seconds, where **Slowbeast** timed out. We point out the significant time advantage of using templates in symbolic execution of programs with loops.

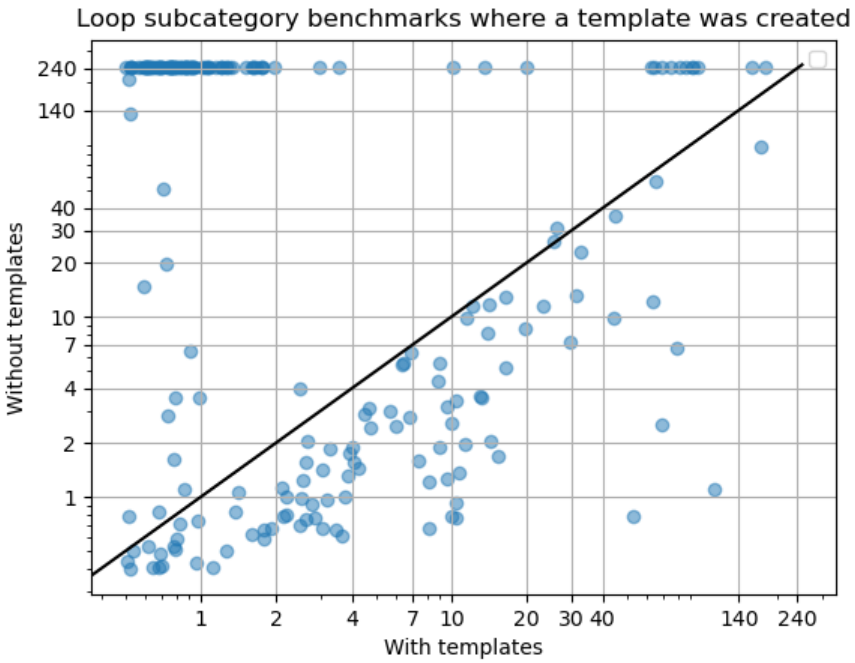

Figure 5.4: Log-scaled scatter plot of 207 benchmarks where compact symbolic execution created a template. The y-axis measures the time taken by **Slowbeast** (without templates), and the x-axis measures the time taken by **Slowbeast CSE** after creating at least one template.

6 Future work

Even in terms of theory, compact symbolic execution has many possible improvements. There may be more effective ways to summarize multiple cyclic paths where we have complete information about their alternation between each other. The code also contains pieces of unfinished prototype functions regarding this issue, which could not be finished in time and their development was stopped, as they are not part of the primary goal of this work.

It may also be worthwhile to eliminate the quantifiers in formulae. One such method is described in the paper of Strejček, Trtík [3]. It will weaken the formula, but the performance gain may be worth it.

There is also the possibility of including more symbolic variable change summarizations. At the moment, arrays, floats, pointers, and division operations are all unsupported mainly because we have not explored such possibilities in this work.

Another modification that could improve the performance is not precomputing the loops. Instead, watch the currently computed path, and if a new cyclic path is detected, remember the loop header and compute a template. A cyclic path is detected simply by finding a node twice in a path whose execution resulted in the current symbolic state.

7 Conclusion

We presented a technique to mitigate the path explosion problem in symbolic execution using templates that summarize the effect of a cyclic path on the symbolic memory and path condition. Next, we presented our approach for implementing this technique into the symbolic executor Slowbeast and described the essential components of compact symbolic execution in pseudocode. After brief instructions about installation and running the benchmarking tool we used, we compared the results of Slowbeast using compact symbolic execution and unmodified Slowbeast on `SV-COMP` 2024 ReachSafety benchmarks.

The results confirmed our expectations of the technique being able to solve loop-oriented benchmarks significantly faster than classic symbolic execution, as well as several areas where compact symbolic execution might perform less effectively.

Additionally, we proposed several other improvements and areas of research regarding this technique. Furthermore, our implementation of compact symbolic execution in Slowbeast has been integrated into Symbiotic, a framework for program analysis.

Bibliography

1. KING, James C. Symbolic execution and program testing. *Commun. ACM*. 1976, vol. 19, no. 7, pp. 385–394. ISSN 0001-0782. Available from DOI: 10.1145/360248.360252.
2. SLABÝ, Jiří; STREJČEK, Jan; TRTÍK, Marek. Compact Symbolic Execution. In: DANG-VAN, Hung; OGAWA, Mizuhito (eds.). *11th International Symposium on Automated Technology for Verification and Analysis, ATVA 2013* [tištěná verze "print"]. Berlin Heidelberg: Springer, 2013, pp. 193–207. ISBN 978-3-319-02443-1. Available from DOI: http://dx.doi.org/10.1007/978-3-319-02444-8_15.
3. STREJČEK, Jan; TRTÍK, Marek. *Abstracting Path Conditions*. 2016. Available from arXiv: 1112.5671 [cs.SE].
4. YI, Qiuping; YANG, Zijiang; GUO, Shengjian; WANG, Chao; LIU, Jian; ZHAO, Chen. Eliminating Path Redundancy via Post-conditioned Symbolic Execution. *IEEE Transactions on Software Engineering*. 2018, vol. 44, no. 1, pp. 25–43. Available from DOI: 10.1109/TSE.2017.2659751.
5. SEN, Koushik; NECULA, George; GONG, Liang; CHOI, Wontae. MultiSE: multi-path symbolic execution using value summaries. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. Bergamo, Italy: Association for Computing Machinery, 2015, pp. 842–853. ESEC/FSE 2015. ISBN 9781450336758. Available from DOI: 10.1145/2786805.2786830.
6. QIU, Rui; YANG, Guowei; PASAREANU, Corina S.; KHURSHID, Sarfraz. Compositional Symbolic Execution with Memoized Replay. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. 2015, vol. 1, pp. 632–642. Available from DOI: 10.1109/ICSE.2015.79.
7. GODEFROID, Patrice. Compositional dynamic test generation. *SIGPLAN Not.* 2007, vol. 42, no. 1, pp. 47–54. ISSN 0362-1340. Available from DOI: 10.1145/1190215.1190226.

8. GODEFROID, Patrice; LUCHAUP, Daniel. Automatic partial loop summarization in dynamic test generation. In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. Toronto, Ontario, Canada: Association for Computing Machinery, 2011, pp. 23–33. ISSTA '11. ISBN 9781450305624. Available from DOI: 10.1145/2001420.2001424.

9. SAXENA, Prateek; POOSANKAM, Pongsin, et al. Loop-extended symbolic execution on binary programs. In: *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*. Chicago, IL, USA: Association for Computing Machinery, 2009, pp. 225–236. ISSTA '09. ISBN 9781605583389. Available from DOI: 10.1145/1572272.1572299.

10. CADAR, Cristian; DUNBAR, Daniel; ENGLER, Dawson. 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*. San Diego, California: USENIX Association, 2008, pp. 209–224. OSDI'08.

11. JONÁŠ, Martin; KUMOR, Kristián; NOVÁK, Jakub; SEDLÁČEK, Jindřich; TRTÍK, Marek; ZAORAL, Lukáš; AYAZIOVÁ, Paulína; STREJČEK, Jan. Symbiotic 10: Lazy Memory Initialization and Compact Symbolic Execution: (Competition Contribution). In: 2024, pp. 406–411. ISBN 978-3-031-57255-5. Available from DOI: 10.1007/978-3-031-57256-2_29.

12. CHALUPA, Marek; STREJČEK, Jan. Backward Symbolic Execution with Loop Folding. In: DRĂGOI, Cezara; MUKHERJEE, Suvam; NAMJOSHI, Kedar (eds.). *Static Analysis*. Cham: Springer International Publishing, 2021, pp. 49–76. ISBN 978-3-030-88806-0.

13. HECHT, M. S.; ULLMAN, J. D. Characterizations of Reducible Flow Graphs. *J. ACM*. 1974, vol. 21, no. 3, pp. 367–375. ISSN 0004-5411. Available from DOI: 10.1145/321832.321835.

14. DE MOURA, Leonardo; BJØRNER, Nikolaj. Z3: an efficient SMT solver. In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Budapest, Hungary: Springer-Verlag, 2008, pp. 337–340. TACAS'08/ETAPS'08. ISBN 3540787992.

15. LATTNER, Chris; ADVE, Vikram. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. Palo Alto, California: IEEE Computer Society, 2004, p. 75. CGO '04. ISBN 0769521029.

A Attached source code and experiments

The attached file `slowbeastcse-main.zip` contains the source code of our modification of Slowbeast as of May 2024[1]. Additionally, the `experiments.zip` archive contains complete logs from our experiment and tables generated by `table-generator` utility of `benchexec`, for ease of viewing.

1. Also available from https://gitlab.fi.muni.cz/xkumor/slowbeastcse