Comparing Different Approaches for Symbolic Execution
ECE 653 - Project Report

Uditya Laad (20986041) (ulaad@uwaterloo.ca)
Karanjot Singh (20928604) (karanjot@uwaterloo.ca)

University of Waterloo, Waterloo, ON, N2L 3G1, CA

Abstract. Symbolic execution has come a long way since its very introduction. Today, we have many sub-categories of symbolic execution techniques aimed at overcoming the challenges and difficulties associated with Static Symbolic Execution (SSE). In this report, we discuss these challenges and how 4 types of symbolic execution techniques, namely – Static Symbolic Execution (SSE), Selective Symbolic Execution (SeSE), Dynamic Symbolic Execution (DSE) (Breadth-first style, Depth-first style), and Symbolic Backward Execution (SBE) compare against each other. To complement this discussion, we have implemented 3 small execution engines, with WLANG as the artifact, using the same structure we had used for creating a Static Symbolic Execution Engine in one of the assignments. We analyze how the 3 new implementations fare against each other for some sample test cases. In the process, we also discuss the design principles, examples and applications of each of these techniques.

Keywords: Symbolic Execution · Static Symbolic Execution · Dynamic Symbolic Execution · Selective Symbolic Execution · Symbolic Backward Execution

Abbreviations:

SE	Symbolic Execution
SSE	Static Symbolic Execution
DSE	Dynamic Symbolic Execution
SeSE	Selective Symbolic Execution
SBE	Symbolic Backward Execution
SAT	Satisfiable
UNSAT	Unsatisfiable
AST	Abstract Syntax Tree

1 Introduction

Symbolic execution (SE) is one of the most widely used techniques to aid software testing. It helps generate test cases for feasible paths in the program, so that useful program paths can be tested instead of just random testing.[4] Symbolic execution traverses the program in consideration, in terms of symbolic expressions, so that multiple feasible paths can be found and respective test cases or satisfying assignments generated.

Why Symbolic Execution? Symbolic execution was introduced in the mid 70's[2] to combat the challenges associated with finding good and meaningful test cases for programs under test. Even though we have other techniques such as random testing, fuzzing, etc; majority of those techniques tend to under-approximate the feasible program properties [2]. Also, some of those techniques, tend to find cases which are either infeasible or take the same execution path as the other test cases (i.e., path duplication). We further discuss how symbolic execution helps overcome these problems so that each new test case is guaranteed to cover a unique path.

Key Idea The main characteristic of symbolic execution is that it maintains the explored path in terms of symbolic expressions, such that all feasible combinations of non-concretized variables can be taken into consideration when deciding whether a particular path can be taken, instead of deciding on the basis of random concrete value for the variables, as is the case with most other random testing techniques. A state in symbolic execution is maintained as a tuple – [2]

$$\begin{aligned} &\textit{either} && < \textbf{path-condition, symbolic-environment} > \\ &\textit{or} && < \textbf{path-condition, symbolic-environment, concrete-environment} > \end{aligned} \tag{1}$$

where 'path-condition' is a first-order Boolean formula that checks for path satisfiability and is updated each time a branch-split (if-then-else, while-do, case, etc), assertion or assumption is encountered. 'Symbolic-memory-mapping or environment' maps each defined variable to its symbolic counterpart. Similarly, 'Concrete-memory-mapping or environment' maps each defined variable to a concrete assignment, such that the assignments satisfy the path-condition for that state.

Checking path satisfiability and finding the corresponding model (i.e., the satisfying assignment) is mostly done using a 'Satisfiability-Modulo-Theory (SMT)' solver. One of the most popular ones in use today in markets is z3-solver by Microsoft [6]

First-hand results Overall, performing symbolic execution on a program, results in an acyclic directed control flow graph, such that each of the leaf nodes have a satisfiable path condition - with a model, to be used as one of the test cases. [4]

We further discuss the many design principles, associated challenges, search strategies and types of symbolic execution engines in detail. Additionally, small implementations of the 3 symbolic execution engines can be found in the project directory, the results of which have been discussed towards the end of this report.

2 How Symbolic Execution finds bugs?

As discussed in the introduction, Symbolic Execution traverses feasible paths and generates respective satisfying assignments which can be utilized as test cases. When the test suite containing these cases is run, the bugs that exist in the respective path are triggered and hence get recorded in the test results. [7] These are the bugs that are found as a result of testing performed, after symbolic execution.

However, certain bugs can be directly found while we are performing symbolic execution. This requires some extra effort. Some oracles such as assertions are compiled into conditionals such that if the path condition and assertions are together VALID, then there are no bugs. Otherwise, even if there is one case, where the combined boolean formula may not hold; then it can be treated as a potential bug in the program. [7]

$$\begin{aligned} &\textit{In general, } \textbf{'assert (condition)'} \textit{ can be expressed as} \\ &\textbf{'if not(condition) then raiseError else proceed'} \end{aligned} \tag{2}$$

The assertions can be used for a safety property such as divide by zero, array index out of bounds, nullPointers, etc.; where the implementation can be explicit (provide assertions or checks before the safe operation) or implicit (added at run-time by the symbolic execution engine – which requires the engine to be created/implemented with that aspect in mind.) [7]. Note that, not all execution engines may support the implicit implementation.

3 Common Properties of Symbolic Execution Techniques

Any version of Symbolic execution tends to share the following common properties [2]:

1. **Progress:** For every CPU resource and second of time that is consumed by the Symbolic Execution Engine, it is justified by progress in terms of – new paths, update to memory stores, removal of infeasible paths or reporting assertion failure. [2]
2. **Uniqueness:** Every new path that is explored should be different from the rest of the paths that have been explored to that point.
3. **Symbolic update:** Symbolic expressions are updated (if applicable) at each step of the program that changes the state; irrespective of whether that part of the program is explored concretely or symbolically.
4. **Maintain History in Path Condition:** 'Path condition' at each new step in the execution, must contain all the constraints that had been explored to reach that node.

4 Common Challenges with Symbolic Execution

Symbolic execution is a very exhaustive process; and even though it produces very good results for the sample programs that we discuss in this report, the same may not hold for programs of bigger size. Following are some of the common challenges faced with all types of symbolic execution techniques, when used with industry and other real-world applications:

Memory Aliasing Having multiple reference variables pointing to the same address in memory, is something that can be very difficult for an SMT solver to manage. However, SE uses symbolic expressions for the variables and mostly does not take dynamic memory allocation into account. [5]

Collections Real-world applications tend to use a lot of collections such as arrays, lists, maps, sets, etc. In such cases, the Symbolic Execution Engine has to make a choice between using a single v/s multiple expressions - to reference the elements of the collection. And a single collection element can be only be referenced dynamically for a concrete iterator [5].

Constraint Satisfiability Satisfiability is an NP complete problem. Hence, the time taken by an SMT solver at each branch point, to determine satisfiability, could be exponential.

Environment Most real-world applications, specially the IoT ones, tend to have a high amount of interaction with the environment or operating system. In such cases, the SE can run into consistency issues (for example, when a system call is made, an SE would no longer have control over the execution). [5]

These challenges are addressed differently in different situations and depend a lot on the context it is used in. While addressing them, SE may have to give up on completeness; but proper assumptions, filters, etc (specific to the context) can be figured out to ensure that the best possible subset of feasible paths are returned by the symbolic execution engine – for the concerned resource and time constraints.

5 Types of Symbolic Execution Engines in focus

5.1 Static Symbolic Execution (SSE)

Static symbolic execution is the same as Classic Symbolic execution that we discussed in **Section-1**. The only point to consider is that the state is a tuple of the first kind:

$$< \textbf{path-condition, symbolic-environment} > \tag{3}$$

Algorithm

Algorithm 1: Static Symbolic Execution - **static_symbolic_execution (current_node, curr _state)**

Input: current_node: Abstract Syntax Tree of the program/block under test
curr_state: Current state in the form of a tuple (path_condition, symbolic_environment).
Output: List of feasible states

```
list_feasible_states = perform_node_specific_execution (current_node, curr_state,
 selected_ast_node, execution_type)
if current_node.has_next() then
    new_states = [ ]
    for state in list_ feasible _states: do
        results = static_symbolic_execution (current_node -> next(), curr_state)
        new_states.extend(results)
    end
    return new_states
end
else
    return list_feasible_states
end
```

Here, the method **'perform_node_specific_execution()'** performs the execution for that specific node (if-then-else, while, assert, assume, assignment, etc) and may also involve a call to **'static_symbolic_execution()'** (if the specific node involves a block execution).

- The implementation of this algorithm can be found in **'wlang/SSE.py'**. (and is the same one that had been submitted for Assignment 2. We use it here only for comparison purposes, against the implementations of Selective Symbolic Execution (SeSE) and Dynamic Symbolic Execution (DSE) engines, which have been freshly implemented).
- The examples/tests discussed in this report can be found in **'wlang/test_SSE_DSE_SeSE.py'**.

Benefits

1. **SSE is complete** - By completeness it is meant that all feasible unsafe assignments are guaranteed to be found using symbolic execution, that is 'SE prevents false negatives'[2].
2. **SSE is sound** - By soundness it is meant that the assignments, which are thought to be unsafe, will actually be realised as unsafe with an error when the assignment is used in a test case, that is 'SE prevents false positives'. [2]

Drawbacks

1. **Path Explosion** [2][5] Real world and industry applications have large amounts of code, and symbolically enumerating each feasible path is not practical most of the times, considering the amount of CPU resources & time at hand
   - The number of paths grow exponentially when each branch point is reached.

- Some loops whose iteration depends on inputs to program, may be caught in an infinite loop.
- Procedures, parallel behaviour, etc – all add to the complexity. [7]
- Potential Solutions: Selective Symbolic Execution

2. **Complex Operations or code:** Operations or code which involve function pointers, hash functions, system calls, etc; can be very hard to execute symbolically [7]. A path condition involving such operations is very difficult for an SMT solver to check
   - Potential Solutions: Dynamic Symbolic Execution
3. **Constraint Satisfiability** Satisfiability is an NP complete problem. Hence, the time taken by an SMT solver at each branch point, to determine satisfiability, could be exponential.

Applications

1. Ideal for small program, which achieve baseline functionality.
2. Ideal in situations, where you have different small modules in an application with very low dependency.

Example

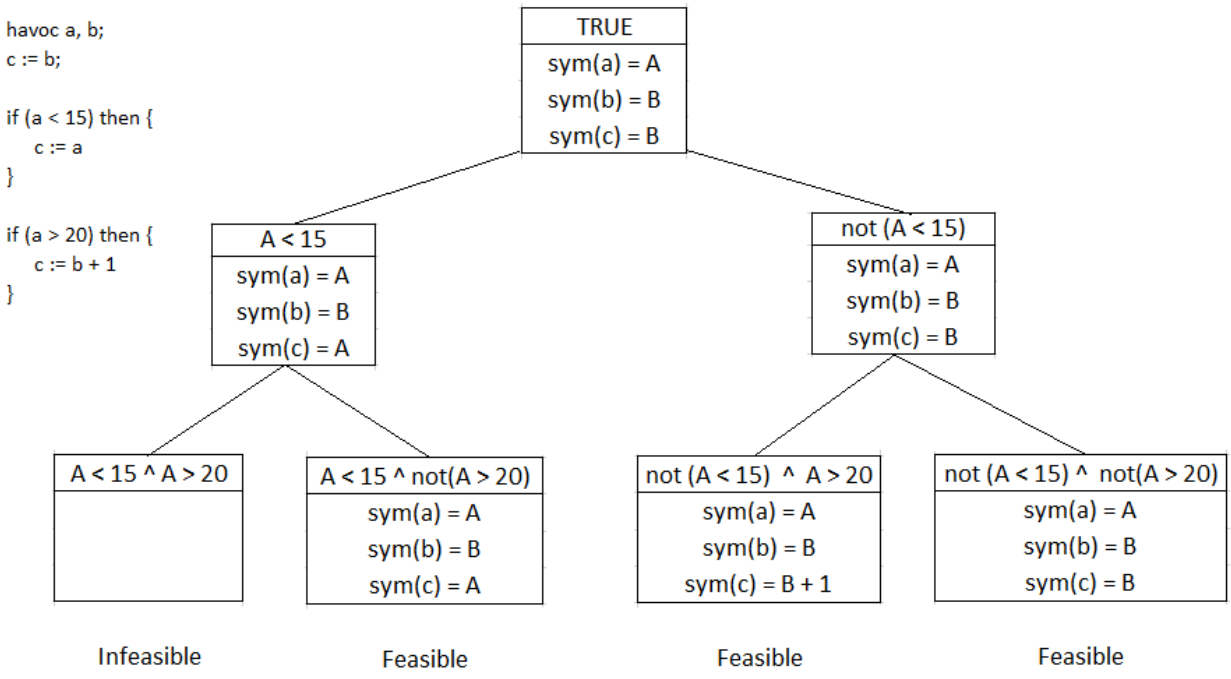

Fig. 1. Example - Static Symbolic Execution (SSE)

5.2 Dynamic Symbolic Execution (DSE)

Dynamic Symbolic Execution is an approach based on concolic execution, which combines both symbolic execution and concrete execution to help guide the engine, when the branch conditions involve operations that are difficult or practically not possible for an SMT solver to check against. Concrete execution essentially drives Symbolic Execution. [2].

A state in Dynamic Symbolic Execution takes the following form:

$$< \textbf{path-condition, symbolic-environment, concrete-environment} > \quad (4)$$

Every time a branch condition is reached, one of the branch is executed concretely by checking if the path condition holds for concrete values of variables. This technique was introduced to overcome the problems associated with static symbolic execution with respect to complex operations such as hash algorithms like sha256 which produces outputs that are extremely difficult (next to impossible) to find a satisfying assignment for. It also helped avoid using the solver for 1 branch condition (although it may lead to incompleteness).

Today, many different execution engines follow a dynamic approach with different search strategies, with Depth-first search being the most commonly used one.

Following are the search strategies that can or are employed in Dynamic Symbolic Execution:

1. **Depth-first Style Dynamic Symbolic Execution**
   Depth first search (DFS) was one of the first search strategies to be used in Dynamic Symbolic Execution. [1] This strategy makes use of a memory-based model, to track back to the last branch condition. The general steps followed in DFS-based DSE are as follows:

Algorithm 2: General Steps for Depth-first Style Dynamic Symbolic Execution

Input: Program AST: Abstract Syntax Tree of the program
curr_state: Current state in the form of a tuple (path_condition, symbolic_environment, concrete_environment).
Output: List of feasible states

(a) Get or find concrete environment values.
(b) Concretely keep executing one of the branches, every time a branch condition is reached. Keep doing so, until either – both the branch conditions are infeasible (with concrete values) or until the leaf node has been reached.
(c) Check the last visited branch in memory.
(d) If the other branch condition involves any complex operations, then compute and replace them concretely in the branch condition. Also, concretize all variables involved in performing the complex operations.
(e) Symbolically check if Path-condition (PC) is SATISFIABLE, using an SMT solver.
(f) If PC is SATIFIABLE, then execute the node, else go to step (c).
(g) Repeat from step (b), until no more branches are left to be covered.

It should be noted that the symbolic environment and path condition is always updated with symbolic expressions at each step, irrespective of whether the branch was executed concretely or symbolically. A related example has been covered in **Section-6.2** of implementation.

Benefits

(a) DFS based DSE helps solve the issue associated with complex operations and functions, making constraint solving easier for that branch.
(b) The number of paths no longer increase in exponential size. Hence, this helps reduce path explosion to some extent.

Drawbacks

(a) Loses completeness: May not be able to cover all feasible paths.
(b) May get trapped in loops, where the condition does not rely on symbolic inputs. [1]
(c) Size of constraint also increases with growth in execution path, and can potentially make the job of constraint solver harder.

Examples of real engines in the industry

(a) DART (Directed Automated Random Testing) [8]
(b) CUTE (Concolic Unit Testing Engine) [9]

2. **Breadth-first Style Dynamic Symbolic Execution**

Breadth first search (BFS), although not as widely used as DFS, is still utilized by some symbolic execution engines. In BFS, the concrete and symbolic branches are executed in parallel, instead of entirely exploring a particular branch (as in DFS).[1]

The general steps followed in BFS-based Dynamic Symbolic Execution are as follows:

Algorithm 3: General Steps for Breadth-first Style Dynamic Symbolic Execution

Input: Program AST: Abstract Syntax Tree of the program
curr_state: Current state in the form of a tuple (path_condition, symbolic_environment, concrete_environment).
Output: List of feasible states

(a) Get or find concrete environment values.
(b) Concretely execute one branch, whichever is feasible with concrete values.
(c) If the other branch condition involves any complex operations, then compute and replace them concretely in the branch condition. Also, concretize all variables involved in performing the complex operations.
(d) In parallel, symbolically execute the other branch.
  - Symbolically check if Path-condition (PC) is SATISFIABLE, using an SMT solver.
  - If PC is SATIFIABLE, then execute the node, else treat the path as infeasible (hence, not explored any further).
(e) Repeat steps (b) to (d) for all the feasible paths in memory – and execute them all in parallel

Note: The symbolic environment and path condition is always updated with symbolic expressions at each step, irrespective of whether the branch was executed concretely or symbolically.

Benefits

(a) Helps solve the issue associated with complex operation such as cryptographic functions.
(b) Can use the multiprocessor capability of modern systems to traverse each path in parallel.

Drawbacks

(a) Loses completeness: May not be able to cover all feasible paths.
(b) Memory Utilization is poor: Since DSE has to keep switching the states. [1]
(c) Exploration space of the program grows quickly. [2]

Examples of real Engines in Industry

(a) EXE [10]
(b) KLEE [11]

Example .

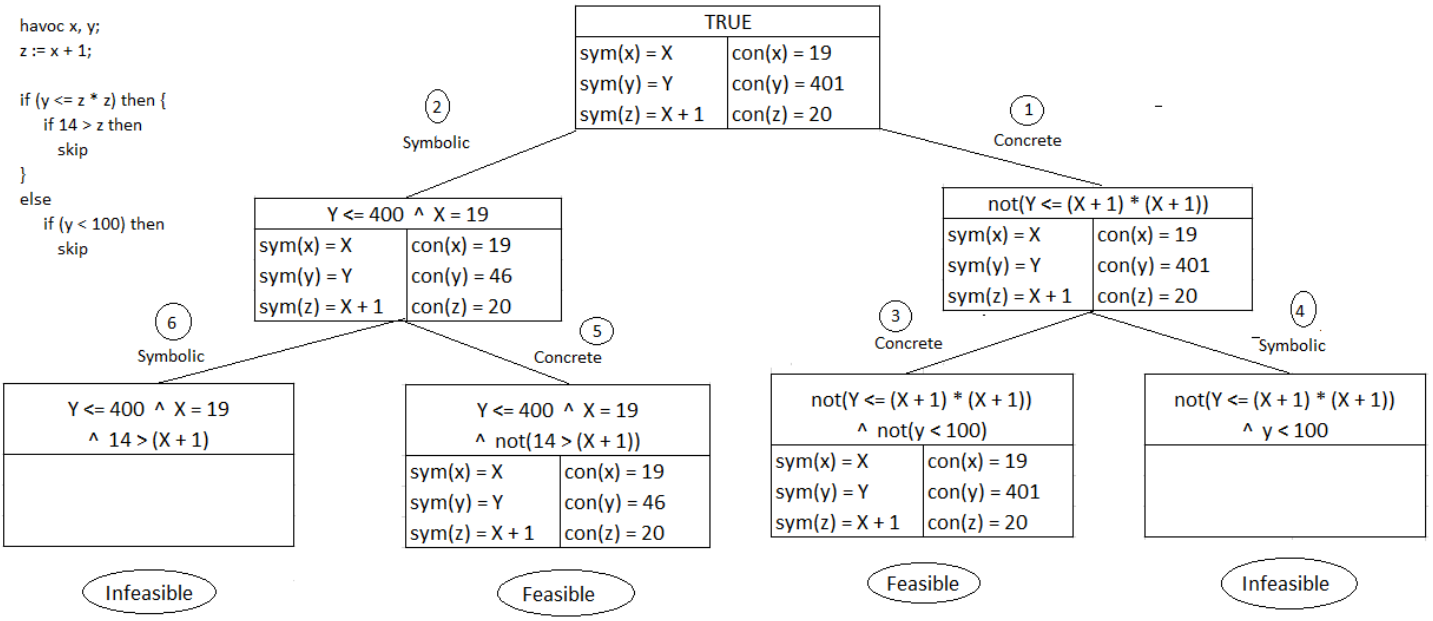

Fig. 2. Example - Breadth-first Style Dynamic Symbolic Execution

3. **Other Search Strategies:** DFS & BFS are classical search strategies that do a full traversal of the execution tree of a program but they can be slow for a huge search space.

   (a) **Random Search:** In this search strategy, execution starts at the root of a program's execution tree and then the execution path is randomly chosen at nodes with conditional statements. This method compromises on stability and does not provide a reliable path coverage but is a quick method to access depth of a program's execution tree in DSE. Hybrid Concolic testing [15] derives from random search but combines it with concolic execution in order to rapidly increase path coverage and thus performing better overall compared to pure DFS or random search.

   (b) **Heuristic Search Strategies:** Heuristic search strategies aim to solve the issue by trading completeness for speed. They use program context to achieve a fast sub-optimal search space by doing the following: [1]
   - Obtain information about the context.
   - Select the coverage metric.
   - Compute the weight associated with each feasible path.

   Some of the widely used Heuristic Search strategies in dynamic symbolic execution include
   - **Generational Search** which uses ExpandExecution as cost function which is based on path constraints and code coverage criteria.
   - **CarFast** which analyses CFG of the program and gives priority to branches containing more states. It is less efficient in DSE when program scale is huge.
   - **Control-Flow Directed Search**: starts by creating a CFG of the program and then calculates and assigns weight of each edge. It then selects least weighted path for DSE.
   - **Fitness-Guided Search**: solves a fitness function to determine the path to be taken.
   - **Context-Guided Search** which, in order to achieve better branch coverage, uses k-context to choose a branch which has a new context for DSE.

5.3 Selective Symbolic Execution (SeSE)

Sometimes, a developer or team may want to test only a part of the program. In such cases, it's not useful to run symbolic execution on the entire program. All of the Symbolic execution techniques which we have discussed so far, do not allow for this option. What we want in this case, is a SE engine which takes input from the developer, as to which part of the program he/she intends to perform testing on. [2] And depending on this input, the engine performs symbolic execution on only the specified part of the program and returns feasible test cases for the respective paths that have been found. The rest of the program can be traversed concretely.[3]

A state in Selective Symbolic Execution takes the following form:

$$< \textbf{path-condition, symbolic-environment, concrete-environment} > \quad (5)$$

SeSE creates an illusion of full symbolic execution [3]. An important factor to consider here is – "once we encounter and finish performing symbolic execution on the module or part of interest, do we continue symbolically executing the rest of the program, or should we switch back to concrete execution?"

We will consider both the cases:

– **The former (Version 1)** makes sense as the code encountered after the part in contention, would have also been affected by whatever changes were made to that part (perhaps as part of maintenance or update). A proper way to implement this case would be as follows:

Algorithm 4: Selective Symbolic Execution - General Steps for Version 1

Input: current_node: Abstract Syntax Tree of the program/block under test
curr_state: Current state in the form of a tuple (path_condition, symbolic_environment, concrete_environment).
Output: List of feasible states

1. Take concrete input and module/part of interest from the developer/user.
2. Run SeSE engine concretely till the module_of_interest has been encountered.
3. Once module_of_interest is encountered, execute the rest of the program symbolically.

– **The latter (Version 2)** also makes sense in some cases, where the code in contention is meant for specific functionality and has no affect on the rest of the program. A proper way to implement this case would be as follows:

Algorithm 5: Selective Symbolic Execution - General Steps for Version 1

Input: current_node: Abstract Syntax Tree of the program/block under test
curr_state: Current state in the form of a tuple (path_condition, symbolic_environment, concrete_environment).
Output: List of feasible states

1. Take concrete input and module/part of interest from the developer/user.
2. Run SeSE engine concretely till the module_of_interest has been encountered.
3. Once module_of_interest is encountered, execute the entire module_of_interest symbolically.
4. Once the module_of_interest has been fully traversed, execute the rest of the program concretely.

Note:

1. In the latter case, switching back and forth between symbolic and concrete execution can be a challenging task. We have successfully implemented both the cases with WLANG as the artifact. The results of our implementation have been discussed in detail in **Section-6.1** and **Section-7**.
2. In Version 1 - the symbolic environment and path condition is always updated with symbolic expressions at each step, irrespective of whether the branch was executed concretely or symbolically. However, in Version 2, we can stop maintaining the symbolic environment, once the module_of_interest has been fully traversed.

Another major challenge in both the cases is – "how to provide the module/part/code of interest". There are many different ways to do this. One of the most effective one is to provide the node in AST (Abstract Syntax Tree), corresponding to that module. And this is exactly the approach that we have followed in our implementations of the 2 versions.

Examples
- Examples can be found in **Section-6.1** and **Section-7**

Benefits

1. Helps reduce the issues associated with path explosion.
2. Time associated with constrained solving in now reduced.

Drawbacks
- Since the part before module_of_interest is executed concretely, there is a strong possibility that the module may never be reached. (This can be taken care of, if the developer provides good concrete values instead of using random ones.)

5.4 Symbolic Backward Execution (SBE)

Symbolic Backward Execution, as the name suggests, is a reverse implementation of the traditional Dynamic Symbolic Execution (DSE). It is used to target a specific line of code such as a throw or assert statement in the code which executes in rare cases on very specific input. It helps identify a test suite that can reach that particular line of code and therefore, is an asset for developers while regression testing and debugging.

The traversal begins from the leaf node (which is the target) and the constraints of the path condition are collected along the branches in the reverse direction. In SBE, many paths can be traversed at a time. Feasibility is checked periodically, such that unsatisfiable paths are discarded and SBE backtracks. [2]

Requirements of SBE: Inter-procedural CFG which helps identify call sites for methods in a program should be made available so that the backward traversal is possible [2]. But even then, it is very difficult to put this into practice.

Benefits

1. SBE can help find a test case that can trigger a specific line in the program.
2. Multiple paths can be traversed in parallel during SBE. [2]
3. Program execution can be traced back from a target line of code to any of the program entry points. [16]

As we can see, Symbolic Backward Execution is completely different technique, compared to the ones we've studied so far, in that they go backwards from Leaf nodes.

Drawbacks

1. Extremely difficult to put into practice since constructing an execution graph with all call sites of a method can be very difficult to construct.
2. Can be very expensive since a method can have many call sites. [2]

6 Small Implementations of 3 Symbolic Execution Engines in WLANG

6.1 Selective Symbolic Execution Engine (SeSE)

We have implemented 2 versions of the Selective Symbolic Execution techniques:

1. **Version 1: Concretely executes till the selected block is reached. Symbolically executes the entire code, from the point where the selected block begins**.
   - Selective Symbolic Execution can be used to cover only the part that has been changed in the most recent update.
   - Hence, it is possible that the rest of the execution is also affected due to the changed block. Hence, we symbolically executed the rest of the code, once the selected node or block has been reached.

2. **Version 2: Symbolically executes only the block or node that was selected. Rest of the code, before and after the selected block is executed concretely.**
   - We often come across changes or updates in code, which do not have a great deal of effect on the rest of the implementation. Also, sometimes it might be computationally and/or financially very expensive to symbolically execute the code after the selected/updated block.
   - Hence, in such cases, it is ideal to symbolically execute only the node that has been selected; and leave the rest to be implemented concretely by the CPU (Central Processing Unit) concretely.

Pre-set

- We are using WLANG as the artifact
- Selection of block of code to be tested, is done by reference to 'AST node' of the corresponding block.
- To simplify the implementation, loop invariants have been supressed.
- A state is a tuple (path_condition, symbolic_environment, concrete_environment)
  where:
  - path_condition: List of contraints in symbolic form
  - symbolic_environment: A dictionary which maps each defined variable to its symbolic value.
  - concrete_environment: A dictionary which maps each defined variable to its concrete value.

Algorithms (Version 1 v/s Version 2)

1. **Version 1:**

Algorithm 6: Selective Symbolic Execution (Version 1) - **version1_selective_symbolic_execution (current_node, curr_state, selected_ast_node, execution_type))**

Input: current_node: Abstract Syntax Tree of the program/block under test
curr_state: Current state in the form of a tuple (path_condition, symbolic_environment, concrete_environment). Initially (in the first recursive call) all 3 are empty/NULL.
selected_ast_node: Node to be executed symbolically
execution_type: Determines whether the implementation should proceed symbolically or concretely. Initially (in the first recursive call) the value is 'concrete'.
Output: List of feasible states

```
if reference (current_node) == reference (selected_ast_node) then
    execution_type = 'symbolic'
end

list_feasible_states = perform_node_specific_execution (current_node, curr_state,
 selected_ast_node, execution_type)

if current_node.has_next() then
    new_states = [ ]
    for state in list_ feasible _states do
        results = version1_selective_symbolic_execution (current_node - next(), curr_state,
         selected_ast_node, execution_type)
        new_states.extend(results)
    end

    return new_states
end
else
    return list_feasible_states
end
```

Here, the method **'perform_node_specific_execution()'** performs the execution (symbolic or concrete – as specified in the argument) and may also involve a call to **'selective_symbolic_execution()'**. Note here, that once we find the 'selected_node', the rest of the execution proceeds symbolically.

- The implementation of this algorithm can be found in **'wlang/SeSE_v1.py'**.
- The examples/tests discussed in this report can be found in **'wlang/test_SSE_DSE_SeSE.py'**.
- Some more interesting test cases can be found in **'wlang/test_SeSE_v1.py'**.

2. **Version 2:**
Here, the method **'perform_node_specific_execution()'** performs the execution (symbolic or concrete – as specified in the argument) and may also involve a call to **'selective_symbolic_execution()'** (for any of the nested nodes - which are a block). **'perform_node_specific_execution()'** also assigns model values to concrete variables (for feasible paths), when branch nodes are reached.

Algorithm 7: Selective Symbolic Execution (Version 2) - **version2_selective_symbolic_execution (current_node, curr_state, selected_ast_node, execution_type))**

Input: current_node: Abstract Syntax Tree of the program/block under test
curr_state: Current state in the form of a tuple (path_condition, symbolic_environment, concrete_environment). Initially (in the first recursive call) all 3 are empty/NULL.
selected_ast_node: Node to be executed symbolically
execution_type: Determines whether the implementation should proceed symbolically or concretely .Initially (in the first recursive call) the value is 'concrete'.
Output: List of feasible states

```
flag_turned = false

if reference (current_node) == reference (selected_ast_node) then
    execution_type = 'symbolic'
    flag_turned = true
end

list_feasible_states = perform_node_specific_execution (current_node, curr_state,
 selected_ast_node, execution_type)

if flag_turned == true then
    execution_type = 'concrete'
end

if current_node.has_next() then
    new_states = [ ]
    for state in list_ feasible _states do
        results = version2_selective_symbolic_execution (current_node - next(), curr_state,
         selected_ast_node, execution_type)
        new_states.extend(results)
    end

    return new_states
end
else
    return list_feasible_states
end
```

Note here, that once we find the 'selected_node', only the 'node-specific execution' is performed fully symbolically.
And the rest of the implementation proceeds concretely.

- The implementation of this algorithm can be found in **'wlang/SeSE_v2.py'**.
- The examples/tests discussed in this report can be found in **'wlang/test_SSE_DSE_SeSE.py'**.
- Some more interesting test cases can be found in **'wlang/test_SeSE_v2.py'**.

Example

1. **Version 1:** Refer Fig 3
2. **Version 2:** Refer Fig 4

Note that: in both cases - there is a possibility that the selected node may not be reached.

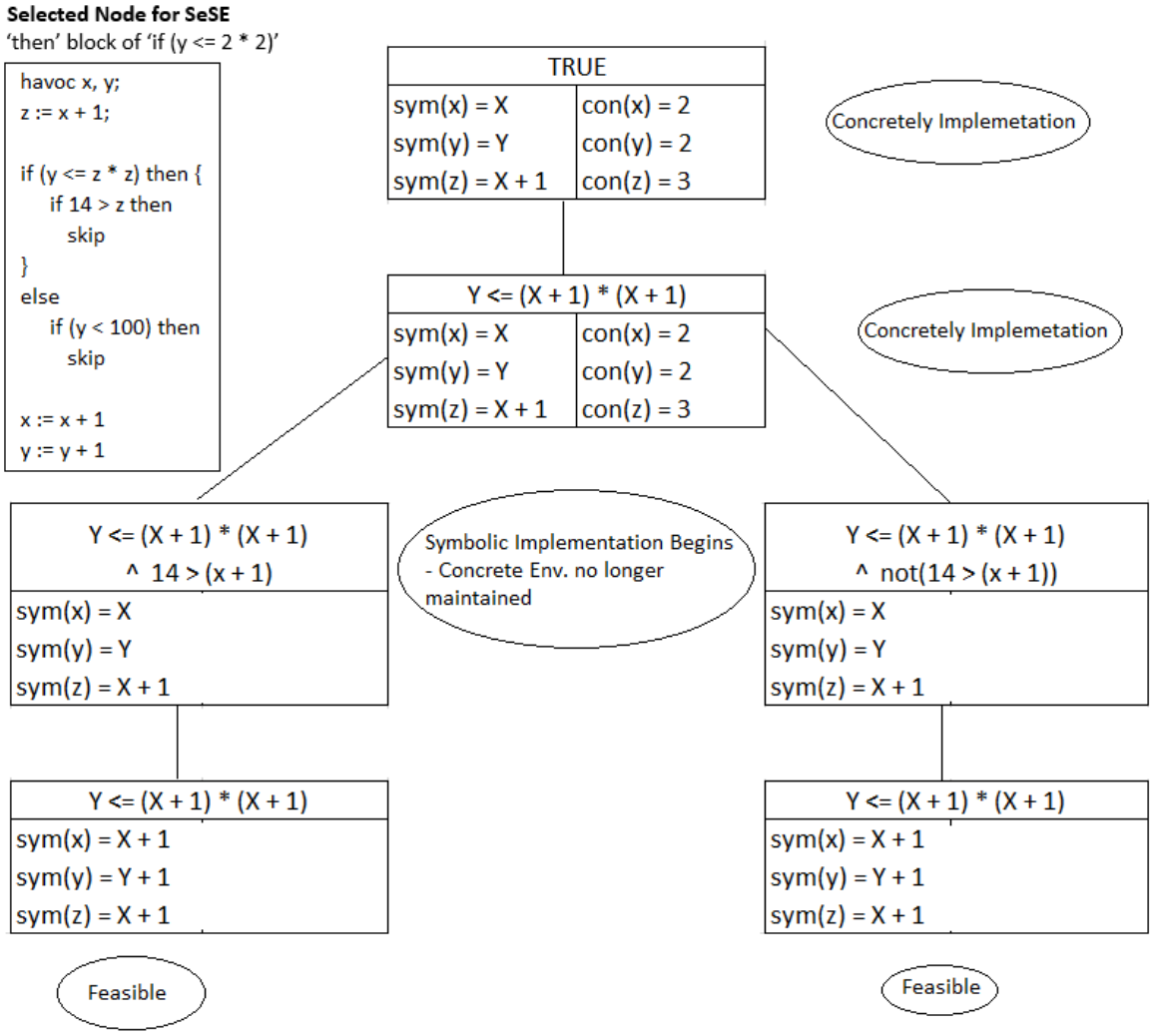

Fig. 3. Example - Selective Symbolic Execution (SSE) - Version 1

6.2 Depth-first-style Dynamic Symbolic Execution Engine

Here, we are implementing a dynamic approach to symbolic execution in a depth-first manner. It is evident that the major benefit of dynamic symbolic execution is the faster processing using CPU for concrete part of the program execution; and in security applications where checking satisfiability of a path condition involving complex operations may be difficult for an SMT solver.

We cover both these aspects in our implementation.

Since we are using WLANG as the artifact and there is no concept of 'methods' in WLANG, for our implementation, we are considering multiplication (*) and division (/) to be complex operations, which most SMT solvers struggle with.

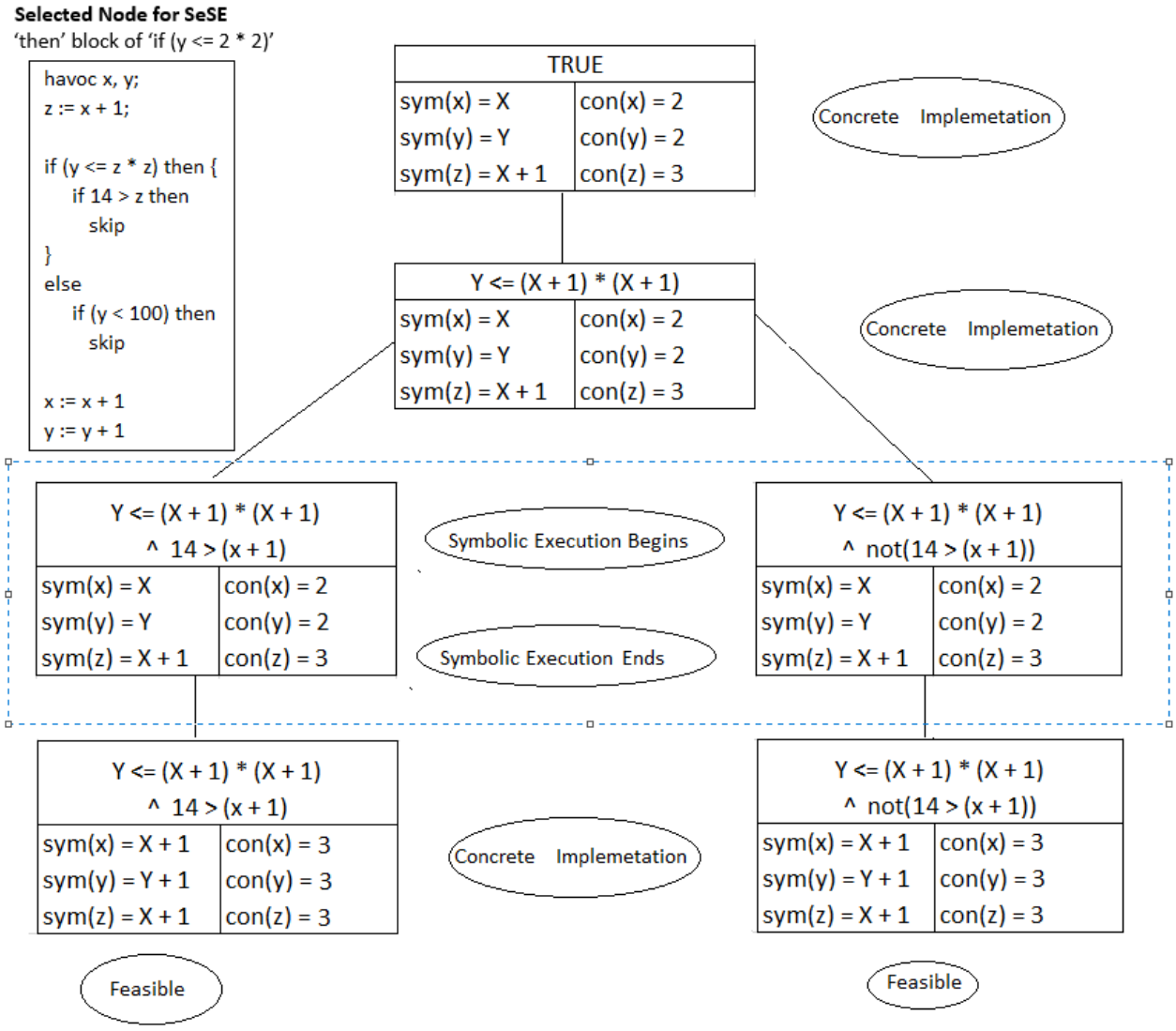

Fig. 4. Example - Selective Symbolic Execution (SSE) - Version 2

Pre-set

- Using WLANG as the artifact
- Complex operations are - multiplication (*) and division (/).
- To simplify the implementation, loop invariants have been supressed.
- A state is a tuple (path_condition, symbolic_environment, concolic_environment)
- Note: For the symbolic branch, we are treating an entire arithmetic expression containing any of the complex operations to be complex expressions, and the symbolic execution proceeds by simplifying the complex expressions using their concrete counterparts.

Algorithm

1. **find_concrete_symbolic():** Finds the branch that can be satisfied with concrete values and assigns the other branch as symbolic.
2. **concretely_perform_node_specific_execution():** Concretely executes branch that holds under the concrete values.
   (a) If any such path exists, then proceeds with that branch-specific execution.

Algorithm 8: Depth-first style Dynamic Symbolic Execution - **dynamic_symbolic_execution (current_node, curr_state)**

Input: current_node: Abstract Syntax Tree of the program/block under test
curr_state: Current state in the form of a tuple (path_condition, symbolic_environment, concolic_environment). Initially (in the first recursive call) all 3 are empty/NULL.
Output: List of feasible states

```
branch_node_type = ['if-then-else', 'while', 'assertion']
list_feasible_states = [ ]

if 'current_node.type' in 'branch_node_type' then
    concrete_branch, symbolic_branch = find_concrete_symbolic (current_node, curr_state)
    result_concrete = concretely_perform_node_specific_execution (concrete_branch,
     curr_state)
    result_symbolic = symbolically_perform_node_specific_execution (symbolic_branch,
     curr_state)
    list_feasible_states.append (result_concrete, result_symbolic)
end
else
    list_feasible_states = perform_node_specific_execution (current_node, curr_state)
end

if current_node.has_next() then
    new_states = [ ]
    for state in list_ feasible _states do
        results = dynamic_symbolic_execution (current_node -> next(), curr_state)
        new_states.extend(results)
    end

    return new_states
end
else
    return list_feasible_states
end
```

3. **symbolically_perform_node_specific_execution():** Symbolically executes the given branch.
   (a) If: 'Path_Condition' involves a complex arithmetic expression;
   - then: a] Compute the expression concretely and directly replace its value in the 'path_condition'
   b] Concretize the symbolic variables, which were involved in the complex arithmetic expression.
   - else: Proceed to 'Step-(b)'
   (b) Check satisfiability.
   (c) If: Satisfiable; then proceed to 'Step-(d)'
   - else: Mark as infeasible and return.
   (d) Get the model from SMT solver; and assign model values to 'concrete variables'.
   (e) Proceed with the branch specific execution.

– The last 2 methods may also involve a call to 'selective_symbolic_execution()' (for any of the nested nodes - which are a block)

– The implementation of this algorithm can be found in **'wlang/DSE.py'**.
– The examples/tests discussed in this report can be found in **'wlang/test_SSE_DSE_SeSE.py'**.
– Some more interesting test cases can be found in **'wlang/test_DSE.py'**.

Example

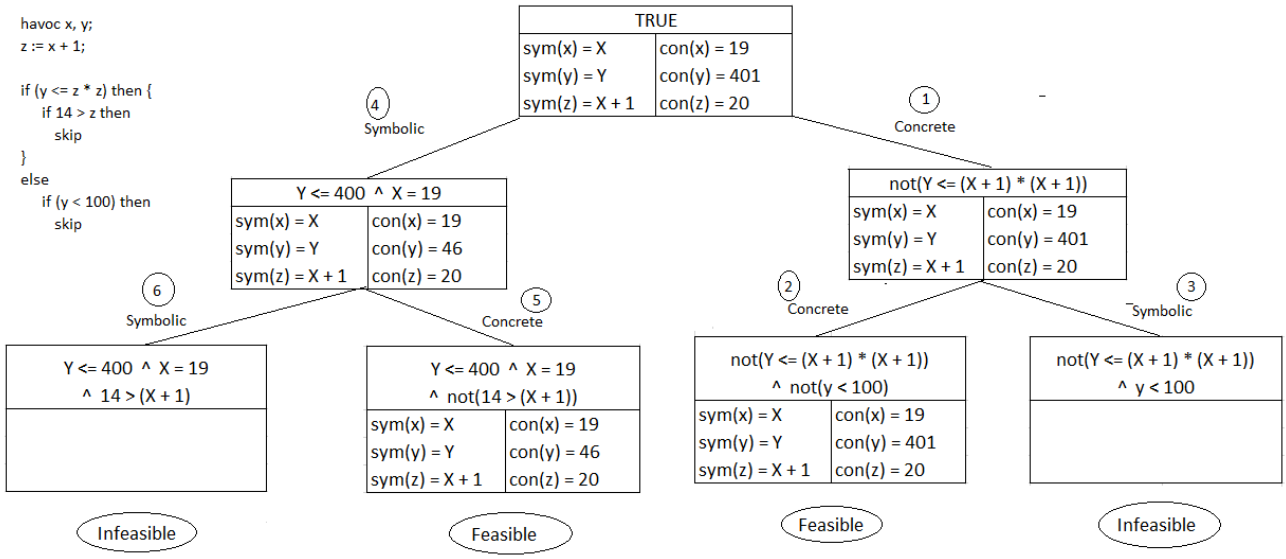

Fig. 5. Example - Depth First Style Dynamic Symbolic Execution

7 Comparing & analyzing the 4 implementations with 3 different sample test cases

As expected, the time taken by Static Symbolic execution is the maximum of the lot. The freshly implemented techniques (DSE, SeSE-1, SeSE-2) in this project are faster than Static SE which was earlier implemented in Assignment 2.

But, its also quite abvious that we miss out on completeness with the new techniques. Each of DSE, SeSE-1, and SeSE-2 return a lower number of feasible paths than their static counterpart.

Another very important aspect that we observe here is that each of the 3 new tehniques can potentially return different number of feasible paths, depending on thier initial concrete values. We had randomized our concrete input for the tests, and hence we see the varying results.

Also; time_taken(SeSE-2) < time_taken(SeSE-1) < time_taken(DSE) in most cases; except for example-1, where we get SeSE-1 taking the lowest-time, but that may be due to the small size and structure of the program. The same condition holds for number of paths returned by these techniques; i.e.: #paths(SeSE-2) < #paths(SeSE-1) < #paths(DSE)

It maybe noted that for DSE, if we run the test cases, and print the feasible states; then we will observe that concrete branch also has a simplified value for the complex sub-expressions. This is essentially because, any symbolic path further down this branch may again need to be concretized to solve for these complex expressions and that too with the same concrete value that the symbolic

path at previous branch was simplififed with (if any). Hence, instead of repeating this procedure, it makes sense to simplify the path for the concrete branch where it first observed that complex expression. This will hold for the concrete split going further and for symbolic split, since we would have had to essentially repeat the same procedure with the same concrete values again, this would also hold for it.

	Example 1		**Example 2**		**Example 3**	

	`havoc x, y;`
	`z := x + 1;`
	
	`if (z*z >= y) then`
	`{`
	`if 14 > z then`
	`skip`
	`}`
	`else`
	`if (y < 100) then`
	`skip`
	
	Selected Node for SeSE (Both Versions):
	'then' block of 'if (z * z >= y)'
			`havoc x, y;`
			`if x + y > 15 then`
			`{ x := x + 7; y := y - 12}`
			`else`
			`{ y := y + 10; x := x - 2 };`
			
			`x := x + 2;`
			`if 2 * (x + y) > 21 then`
			`{`
			`x := x * 3;`
			`y := y * 2`
			`}`
			`else`
			`{`
			`x := x * 4;`
			`y := y * 3 + x`
			`};`
			`skip`
			
			Selected Node for SeSE (Both Versions):
			1st 'if-then-else' block
					`r := 0;`
					`havoc x;`
					
					`if x > 8 then {`
					`havoc x;`
					`r := x - 7`
					`};`
					
					`if x / 1 < 5 then`
					`r := x - 2`
					
					Selected Node for SeSE (Both Versions):
					2nd 'if-then' block	

	Time	**# of feasible paths**	**Time**	**# of feasible paths**	**Time**	**# of feasible paths**
Static Symbolic Execution	0.151s	4	0.152s	3	0.134s	4
Dynamic Symbolic Execution	0.139s	2 or 3	0.170s	2	0.147s	2
Selective Symbolic Execution – Version 1	0.091s	1 or 2	0.142s	3	0.133s	2
Selective Symbolic Execution – Version 2	0.120s	1 or 2	0.121s	2	0.115s	2

Fig. 6. Analyzing the 4 implementations with some sample test cases

8 Conclusion

As we observed through our analysis, we can adopt different techniques of symbolic execution and also in different configurations to tackle the problems associated with Static or Classical symbolic execution, most particular being 'path explosion' and 'complex operations'.

The symbolic execution techniques have been listed in **Fig-6** in decreasing order of their association with symbolic activity. And hence we observe a lower time, required for symbolic processing, while the no. of feasible paths also decrease; from SSE to DSE to SeSE-1 to SeSE-2.

While SeSE may be the fastest techniques of the lot, they are not always practical in real-life. DSE is the most widely used technique, as it helps reduce path explosion and saves computing

resources utilized by the solver; while also maintaining a relatively higher level of symbolic association.

Depending on the case and context, a developer or organization may choose to utilize any of these techniques or may even try and come up with thier own version of symbolic execution.

References

1. Yu LIU*, Xu ZHOUa and Wei-Wei GONGb.: A Survey of Search Strategies in the Dynamic Symbolic Execution https://www.itm-conferences.org/articles/itmconf/abs/2017/04/itmconf_ita2017_03025/itmconf_ita2017_03025.html
2. ROBERTO BALDONI, EMILIO COPPA, DANIELE CONO D'ELIA, CAMIL DEMETRESCU, and IRENE FINOCCHI, Sapienza University of Rome.: A Survey of Symbolic Execution Techniques https://arxiv.org/abs/1610.00502
3. Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, George Candea School of Computer and Communication Sciences ´E cole Polytechnique F ´ed ´erale de Lausanne (EPFL), Switzerland.: Selective Symbolic Execution https://infoscience.epfl.ch/record/139393
4. Tutorials Point, Software testing dictionary, https://www.tutorialspoint.com/software_testing_dictionary/symbolic_execution.htm
5. Wikipedia, Symbolic Execution, https://en.wikipedia.org/wiki/Symbolic_execution
6. Micrisoft Research, z3, https://www.microsoft.com/en-us/research/project/z3-3/
7. University of Waterloo, ECE 653, Symbolic Execution, https://git.uwaterloo.ca/stqam-1225/pdfs/-/raw/master/W05-SymExec.pdf
8. Patrice Godefroid Nils Klarlund, Koushik Sen.: DART: Directed Automated Random Testing, https://web.eecs.umich.edu/~weimerw/590/reading/p213-godefroid.pdf
9. Koushik Sen, Darko Marinov, Gul Agha.: CUTE: A Concolic Unit Testing Engine for C https://mir.cs.illinois.edu/marinov/publications/SenETAL05CUTE.pdf
10. Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, Dawson R. Engler.: EXE: Automatically Generating Inputs of Death, https://dl.acm.org/doi/10.1145/1455518.1455522
11. KLEE, Github IO, 11.https://klee.github.io/
12. University of Waterloo, ECE 653, Dynamic Symbolic Execution, https://git.uwaterloo.ca/stqam-1225/pdfs/-/raw/master/W06-DSE.pdf
13. Patrice Godefroid, Michael Y. Levin, David Molnar, Microsoft.: SAGE: Whitebox Fuzzing for Security Testing , https://queue.acm.org/detail.cfm?id=2094081
14. Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks: Directed symbolic execution, https://www.cs.umd.edu/~mwh/papers/dse-sas11.pdf
15. Majumdar, Rupak, and Koushik Sen. "Hybrid concolic testing." 29th International Conference on Software Engineering (ICSE'07). IEEE, 2007.
16. Peter Dinges and Gul Agha. 2014. Targeted test input generation using symbolic-concrete backward execution. In Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE'14). 31–36. DOI, http://dx.doi.org/10.1145/2642937.2642951