TODO Add a name

Kevin Chen

February 26, 2026

1 Introduction

TODO insert introduction

2 Symbolic Execution

Symbolic execution is a way of executing a program abstractly, such that on abstract execution covers multiple possible inputs of the program that share a particular execution path through the code [1]. Each input of symbolicly executed code is represented as a symbol rather than assigning it a concrete value. When the value is referenced such as for an if statement, the execution path branches, where one path the symbolic value is evaluated as true while the other path the symboilic value is evaluated as false.

One of the key uses for symbolic execution is in testing. Most unit tests, require the programmer to hardcode in values to test, resulting in various unit tests to cover all branches the code can tranverse. On the other hand, a symbolic execution would execute simutaneously on a family of inputs. Consider the following example:

Algorithm 1 Algorithm 1

Require: $x, y \in \mathbb{Z}$

1: $a \leftarrow x + y$
2: **if** $a \bmod 2 = 0$ **then**
3: **return** error "a is even"
4: **else**
5: **return** a
6: **end if**

With a glance, we can see that using any combination of numbers such that their sum results in an even number will result in an error. There is no way to cover all cases of failure using unit tests, as there are an infinite number of way which we can stumble onto this error.

Instead of testing with unit tests such as $x = 1, y = 1$, symbolic execution represents the inputs as symbols $x = x_0, y = y_0$. Whenever, the symbols are required to return a value, such as in a branch, symbolic execution splits into two executions, with each execution recording the branching condition in its execution path. Consider a symbolic execution of the algorithm above. We start with representing each input symbolicly:

line	Path Condition	Translation
0	true	$x \to x_0, y \to y_0$
1	true	$x \to x_0, y \to y_0, a \to x_0 + y_0$
2	$(x_0 + y_0) \bmod 2 = 0$	$x \to x_0, y \to y_0, a \to x_0 + y_0$
2	$\neg((x_0 + y_0) \bmod 2 = 0)$	$x \to x_0, y \to y_0, a \to x_0 + y_0$
3	$(x_0 + y_0) \bmod 2 = 0$	Error
5	$\neg((x_0 + y_0) \bmod 2 = 0)$	Return $x_0 + y_0$

Figure 1: Symbolic execution trace showing path conditions and variable translations

In this example, we see that the inputs values start symbolic as symbols which has no concrete value. Following the execution, we translate every instance of x with x_0 with the same case for y. Where it gets interesting is when the the execution has a branching path. In the **if** statement, the value of a is required to be assigned a value, which breaks the symbolic abstraction. Instead, the symbolic execution branches into two different executions, each of which assuming a different condition to satisfy the branching statement. Below we can visualize the different execution paths the execution takes via a simple tree (see Figure 2):

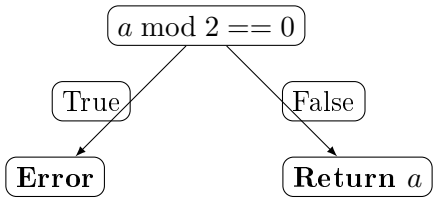

Figure 2: Execution tree showing the branching paths in symbolic execution

In each paths, we accumulate the different path conditions until we either reach the end (return statement) or a failure (error). In this case, we find that there exists a set of inputs (when $x_0 + y_0 \bmod 2 = 0$), which causes an error. Now imagine a different example; using the same coding example, but with a twist. Prior to running this program with symbolic execution, one can add constraints to the path. Below, an example where we start with a given constraint that $x_0 + y_0 = 1$:

line	Path Condition	Translation
0	$x_0 + y_0 = 1$	$x \to x_0, y \to y_0$
1	$x_0 + y_0 = 1$	$x \to x_0, y \to y_0, a \to x_0 + y_0$
2	$\neg((x_0 + y_0) \bmod 2 = 0)$	$x \to x_0, y \to y_0, a \to x_0 + y_0$
2	$x_0 + y_0 = 1 \wedge (x_0 + y_0) \bmod 2 = 0$	$x \to x_0, y \to y_0, a \to x_0 + y_0$
3	False	Error
5	$\neg((x_0 + y_0) \bmod 2 = 0)$	Return $x_0 + y_0$

Figure 3: Modified version of execution with existing constraints

With the constraint, the path condition which leads to the error is contradictory. This illustrates that the path is impossible. With this constraint, we see that no path leads to any errors. In industry, programmers, can create tests with given preconditions (constraints) and run until the program terminates. Instead of throwing an error, we can use assertion statements as post-conditions. If it terminates without an error, the program is robust across all set of inputs. Otherwise, a symbolicly executed program will return a concrete set of conditions which will give an error or violate a post-condtiion.

2.1 While loops

To represent a while loop in symbolic execution, we unroll it into a recursive function with no loops. Take the following code for instance:

Algorithm 2 Count Number of 1's in a Bit Vector

Require: v is a bit vector of length n

1: $count \leftarrow 0$
2: $i \leftarrow 0$
3: **while** $i < n$ **do**
4: **if** $v[i] == 1$ **then**
5: $count \leftarrow count + 1$
6: **end if**
7: $i \leftarrow i + 1$
8: **end while**
9: **return** *count*

We can rewrite it as a recursive function:

Algorithm 3 Recursively Count Number of 1's in a Bit Vector

Require: v is a bit vector of length n

1: **function** COUNT_ONES(v, i, n)
2: **if** $i == n$ **then**
3: **return** 0
4: **else if** $v[i] == 1$ **then**
5: **return** 1+ COUNT_ONES($v, i + 1, n$)
6: **else**
7: **return** COUNT_ONES($v, i + 1, n$)
8: **end if**
9: **end function**
10: **return** COUNT_ONES($v, 0, n$)

Now that we have rewritten the while loop as if statements, we simply create a new branch for every if statement. In practice, this may lead to state explosion, resulting in thousands of branches in a just a few iterations. Programmers either set a recursion limit, terminating the branch which gets stuck, or find a different way to represent the while loop, using the loop invariant. Below is a tree of the branching paths created from a symbolic execution of Algorithm 3.

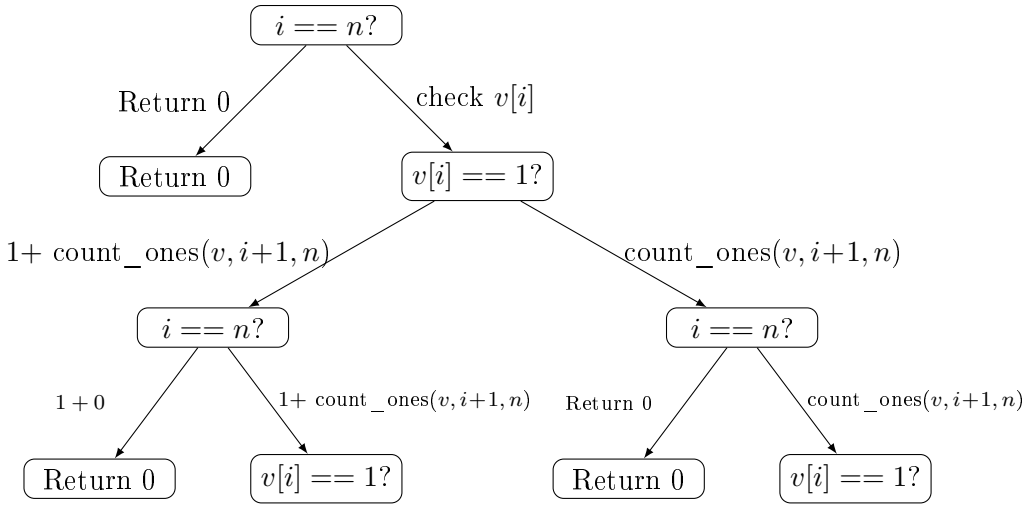

Figure 4: Symbolic execution tree for the recursive count_ones algorithm showing branching at each condition

3 Patronus

3.1 BTOR2

BTOR2 is a word-level model checking format for capturing models of hardware and potentially software in a bit-precise manner. The exact architecture in which it is defined can be found at [3].

For the purposes of this project, we simplify the language into a transition system which holds state variables, inputs, constraints, and bad states. To set up the system, the state variables start by holding some initial value and transition function. To take a step in the system, translate all the state variables through their existing transition functions. Inputs are free, nondeterministic values chosen at each transition, so long as they satisfy all constraints. At the end of the step, check that the bad state predicate is not satisfied.

The components of the transition system map directly to BTOR2 keywords: `state` (with `init` and `next`) for state variables, `input` for free inputs, `constraint` for invariant assumptions on inputs, and `bad` for the safety violation predicate. Algorithm 4 formalizes this execution model; Table 5 traces a concrete two-step run that witnesses a bad state.

Algorithm 4 BTOR2 Bounded Model Checking

Require: Transition system $(\mathcal{S}, \mathcal{I}, \mathit{init}, \mathit{next}, \mathcal{C}, \mathcal{B})$; bound K

Init: for each $s \in \mathcal{S}$, set $s \leftarrow \mathit{init}(s)$
for $k = 0, 1, \ldots, K$ **do**
  if $\mathcal{B}$(state) **then**
    return UNSAFE ▷ bad state reached at step k
  end if
  Choose values for each $i \in \mathcal{I}$ satisfying every $c \in \mathcal{C}$
  for each $s \in \mathcal{S}$ **do**
    $s \leftarrow \mathit{next}(s, \mathcal{I})$
  end for
end for
return UNKNOWN

Consider a system with one state variable s (initialized to 0, $next(s) = s + inp$), one free input inp constrained by $inp \geq 1$, and bad state $\mathcal{B} \equiv s \geq 3$.

Step	**State** (s)	**Bad?** ($s \geq 3$)	**Input** (inp)	**Next State** (s')
Init	0	false	1	1
1	1	false	2	3
2	3	**true**	–	–

Figure 5: Two-step BTOR2 execution trace. State s is initialized to 0; each step applies a chosen input (constrained to ≥ 1) and updates s via *next*. The bad predicate $s \geq 3$ is first satisfied at step 2, witnessing the safety violation.

3.2 SMT Solver

One of the key challenges of implementing symbolic execution is being able to distinguish whether or not a path is feasible or not. How should a symbolic executor tell whether or not a path is feasible such as in Figure 3 row 4 or infeasible, like in Figure 3 row 3.

As always in computer science, one of the best ways to model something is to make it someone else's problem: take an existing solution to a hard problem and leverage it to solve your own. Here, we leverage a SAT solver to determine path feasibility. The remaining challenge is translating each branching condition into a series of boolean equations. This is the role of an SMT (Satisfiability Modulo Theories) solver, which lifts raw SAT solving to richer data types. For example, integers are modeled as fixed-width bit-vectors; a standard 32-bit integer becomes a vector of 32 boolean variables, one per bit.

Consider a 4-bit unsigned integer $x = (x_3, x_2, x_1, x_0)$, where x_3 is the most significant bit. To check the condition $x > 4$, the SMT solver observes that $4 = (0, 1, 0, 0)_2$ and reasons case-by-case over the leading bits: if $x_3 = 1$ then $x \geq 8 > 4$ regardless; if $x_3 = 0$ and $x_2 = 1$ then $x \in \{4, \ldots, 7\}$, and $x > 4$ requires at least one of x_1, x_0 to be 1; otherwise $x \leq 3$. The SMT solver therefore translates $x > 4$ into the propositional formula

$$x_3 \ \vee \ \big(x_2 \ \wedge \ (x_1 \vee x_0)\big)$$

and passes this purely boolean clause to the SAT solver.

3.3 Expressions

Patronus[2] represents expressions under the class *Expr* and *ExprRef*. Each expression is an object which contains the type of expression and a list of *ExprRef* to other expressions. For example, the expression $if(x == 4, y+ = 3, z+ = 4)$ can be represented as $ite(ExprRef[1], ExprRef[2], ExprRef[3])$. The expressions are stored all in an Context.

4 Implementation

4.1 Initialize

In the initiation, each state variable will starts with an symbolic value. However, if the state variable has an initial value, it will be assigned to that instead. For simplicity, values assigned at each timestep will have a timestep added as part of its symbol. This is to distinguish between symbols added and important for identifying different symbols for the input.

4.2 Simplification

The goal of simplification is to have values as simplist terms. This means that each expression must contain basic terms, such as numbers or symbols, things that cannot be further simplified.

Assume you have all the basic rules for simplification. One example is if you add two integers, the result should just be the sum of the integers. Another is if you have an if statement and the condition is true, then it should simplify to just the true branch.

To simplify expressions and expression references, we need to start by simplifying the inner most layer. This is implemented with a stack. Each child of an expression gets added to a stack and the last children added is popped and simplified. Table 6 traces this algorithm on the expression $If(true, If(add(0,1),4,5),-1)$: the first column is the step (number of pops), the second is the current result overall (the full expression with all completed simplifications applied so far), the third is the current simplification target (the expression just popped), and the fourth is the stack (top to bottom) after pushing that target's children, if any. The expression simplifies to 4: -1, 4, and 5 do not simplify; $add(0,1)$ simplifies to 1; $If(1,4,5)$ simplifies to 4 (condition is true); $If(true,4,-1)$ simplifies to 4.

Step	**Current result overall**	**Current simplification target**	**Stack**
0	$If(true, If(add(0,1),4,5),-1)$	$If(true, If(add(0,1),4,5),-1) \rightarrow ?$	$true,\ If(add(0,1),4,5),\ -1$
1	$If(true, If(add(0,1),4,5),-1)$	$-1 \rightarrow -1$	$true,\ If(add(0,1),4,5)$
2	$If(true, If(add(0,1),4,5),-1)$	$If(add(0,1),4,5) \rightarrow ?$	$true,\ add(0,1),\ 4,\ 5$
3	$If(true, If(add(0,1),4,5),-1)$	$5 \rightarrow 5$	$true,\ add(0,1),\ 4$
4	$If(true, If(add(0,1),4,5),-1)$	$4 \rightarrow 4$	$true,\ add(0,1)$
5	$If(true, If(1,4,5),-1)$	$add(0,1) \rightarrow 1$	$true,\ If(1,4,5)$
6	$If(true,4,-1)$	$If(1,4,5) \rightarrow 4$	$true$
7	4	$true \rightarrow true$	$\emptyset$

Figure 6: Stack-based simplification trace for $If(true, If(add(0,1),4,5),-1)$. At each step the target is popped and its children (if any) are pushed. -1, 4, 5, 0, 1, and $true$ do not simplify; $add(0,1) \rightarrow 1$, then $If(1,4,5) \rightarrow 4$, then $If(true,4,-1) \rightarrow 4$. Result: 4.

4.3 Finding nested if-statements

Using the idea for simplification, the same method can be used to traverse through nested if-statement. Start by implementing a stack which holds all the expressions, a stack which holds all the traversed paths, and a map which holds the position of each expression reference. For each expression, add each child to the mapping, and then explore each children. If any of the children are an ITE expression, we should simplify the path to either the true or false expression and add this to the path stack. Then, with the assumption that the ite is true or false, we can use simplification to simplify the expression and add it to the expression stack. Do this for each path as well and eventually we will have a list of paths which represent all feasible combinations of paths and, for each path, the new states.

Consider the expression $add(ite(x,1,2),\ ite(y,2,3))$ where x and y are unconstrained symbolic booleans. There are two nested ITE sub-expressions, one inside each argument of add, producing four feasible execution paths. Figure 7 shows the resulting path-exploration tree.

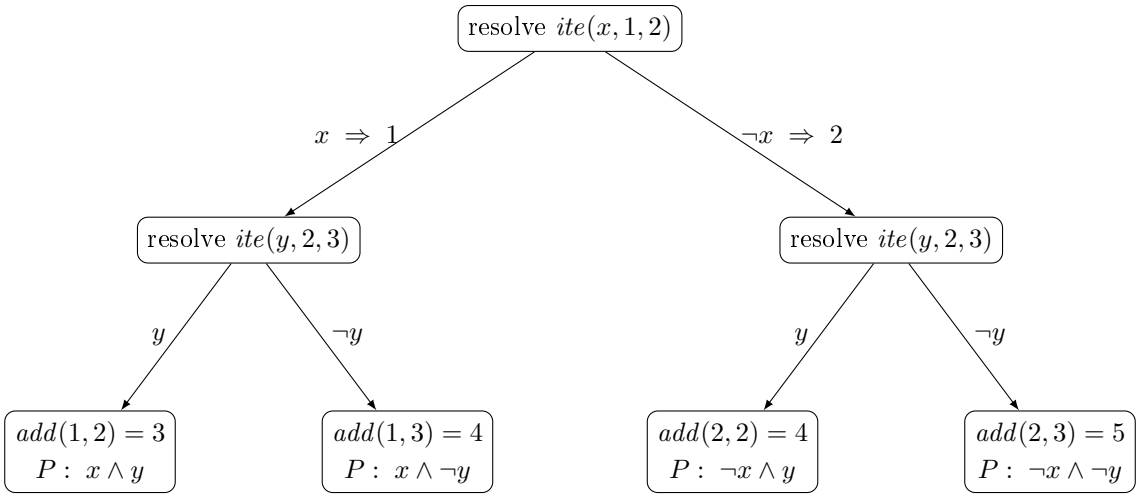

Figure 7: Path-exploration tree for $add(ite(x, 1, 2), ite(y, 2, 3))$. Each internal node shows an ITE being resolved; edges are labelled with the branch condition and its concrete value. Leaves show the fully resolved result and the accumulated path conditions P.

Not every *ite* expression necessarily results in a branch during symbolic execution. Consider the expression $ite(x > 2, ite(x > 1, 0, 1), 1)$: the condition of the inner *ite* is already entailed by the outer condition, rendering one of its branches unreachable. To handle this precisely, we issue the following queries to the SMT solver:

1. $P \wedge c = SAT \wedge P \wedge \neg c = SAT$: The SMT solver says that both paths are feasible. In this case, we create two branching executions one with $P' = P \wedge c$ and one with $P' = P \wedge \neg c$.

2. $P \wedge c = SAT \wedge P \wedge \neg c = UNSAT$: The SMT solver says that the path such that $\neg c$ is impossible. This means that given the information of P, $\neg c$ is infeasible. As such, we do not need to add $\neg c$ into the path, as the information is already recorded in the path. You can think of this as a node with one branch. It has no other choice so no branching executions are created. Consider $P = (x == 10) \wedge (y == 11), c = (x + y > 12)$. There is no need to add c to the execution path because the value is already included in P.

3. $P \wedge c = UNSAT \wedge P \wedge \neg c = SAT$: She SMT solver says that the path such that c is impossible. Similar to the previous case, we do not need to add c onto the path as the information for deducing $\neg c$ is already included in P.

4. $P \wedge c = UNSAT \wedge P \wedge \neg c = UNSAT$: The result states that regardless of c the execution is infeasible. This must mean that P is infeasible, which means the constraints were infeasible.

4.4 Path Merging

The current implementation of symbolic execution with just one *ite* expression will double in size with each step. Consider the following state: $x := ite(i > 0, x + 1, x - 1)$. Given the constraint that i is an input and that i is either 1 or 0, there is exists two branches in each step. As such within 5 steps, the number of symbolic execution paths will be $2^5 = 32$.

To counter state explosion, we imploy is path merging. In the previous example, consider the branch that the input is first 0 then 1 versus first 1 then 0. Both symbolic paths result in the same system state. As a result, we can combine the two paths with an $\wedge$ to prevent state explosion.

Figure 8 illustrates the first two steps of this example (x_0 is the initial value of x). Without merging, the four paths at step 2 are tracked separately. With merging, the two paths that reach $x = x_0$ (one via $i_1{=}0, i_2{=}1$ and one via $i_1{=}1, i_2{=}0$) are combined, reducing the live path count.

Step	i **sequence**	**Value of** x	**Path condition**	**Merged?**
0	—	x_0	$\top$	—
1	$i_1{=}1$	$x_0 + 1$	$i_1 > 0$	No
1	$i_1{=}0$	$x_0 - 1$	$\neg(i_1 > 0)$	No
2	$i_1{=}1,\ i_2{=}1$	$x_0 + 2$	$i_1 > 0 \wedge i_2 > 0$	No
2	$i_1{=}1,\ i_2{=}0$	x_0	$i_1 > 0 \wedge \neg(i_2 > 0)$	} merged
2	$i_1{=}0,\ i_2{=}1$	x_0	$\neg(i_1 > 0) \wedge i_2 > 0$	
2	$i_1{=}0,\ i_2{=}0$	$x_0 - 2$	$\neg(i_1 > 0) \wedge \neg(i_2 > 0)$	No
2 (merged)	$i_1 \neq i_2$	x_0	$(i_1 > 0 \wedge \neg(i_2 > 0)) \vee (\neg(i_1 > 0) \wedge i_2 > 0)$	✓

Figure 8: Path explosion and merging for $x := \mathit{ite}(i{>}0,\ x{+}1,\ x{-}1)$ over two steps. The two step-2 paths that reach x_0 are merge candidates; their path conditions are joined with $\vee$.

5 Conclusion

References

[1] Jonathan Aldrich. Symbolic execution. Lecture notes 14, Course 17-355, Carnegie Mellon University, 2018. Accessed: 2025.

[2] Kevin Laeufer. Patronus. GitHub repository, 2024. Accessed: 2025.

[3] Aina Niemetz, Mathias Preiner, Clifford Wolf, and Armin Biere. BTOR2, BtorMC and Boolector 3.0. In *Computer Aided Verification (CAV)*, volume 10981 of *Lecture Notes in Computer Science*, pages 587–595. Springer, 2018.