Tracer-X: Dynamic Symbolic Execution with Interpolation

Joxan Jaffar, Rasool Maghareh,
Sangharatna Godboley, Xuan-Linh Ha

Department of Computer Science, National University of Singapore
*{joxan,rasool,sanghara,haxl}@comp.nus.edu.sg

January 2020

Introducing *TRACER-X* symbolic execution approach:

- Based on the KLEE symbolic virtual machine
- Utilize *Interpolation* for search-space reduction

Outline:

1. Symbolic Execution
2. TRACER-X (Symbolic Execution with Interpolation)
3. Weakest Precondition Interpolation
4. Conjunctive Path-based Weakest Precondition Interpolation
5. Results

Symbolic Execution

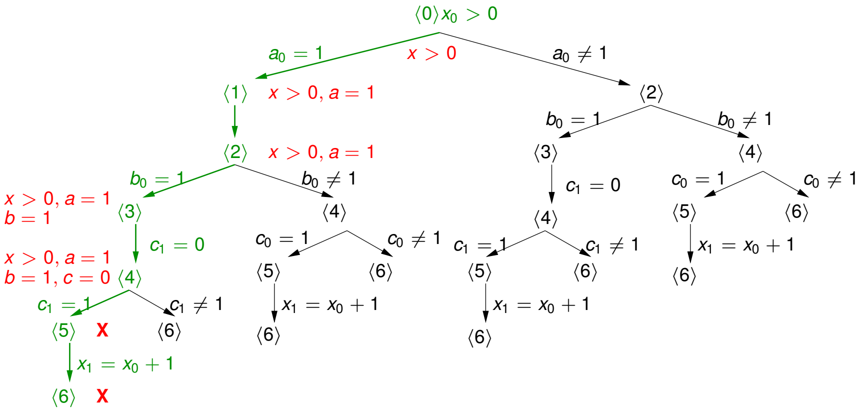
- Model program statements as *constraints*
- Analysis/verification by traversal on symbolic execution tree

Example:

Initially $x > 0$
$\langle 0 \rangle$ **if** $(a = 1)$ **then** $\langle 1 \rangle$ **skip endif**
$\langle 2 \rangle$ **if** $(b = 1)$ **then** $\langle 3 \rangle$ $c := 0$ **endif**
$\langle 4 \rangle$ **if** $(c = 1)$ **then** $\langle 5 \rangle$ $x := x + 1$ **endif** $\langle 6 \rangle$

Safety Property: We want to prove that the execution results in positive x.

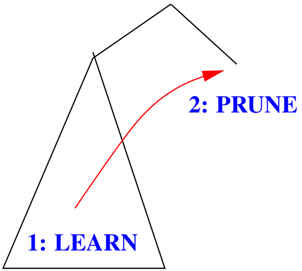
Symbolic Execution Tree



- *Symbolic State:* constraints on program variables

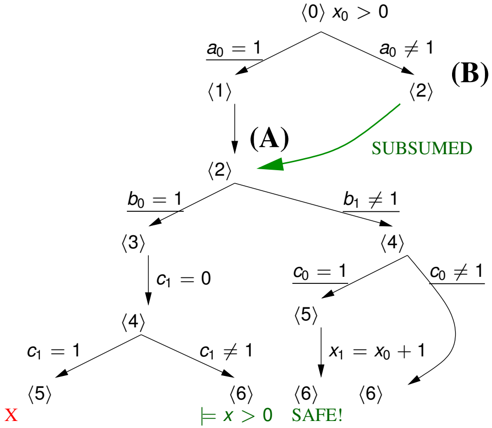
Problem and Solution

- Naive analysis/verification (e.g., standard model checking) $\rightarrow$ huge search space: exponential in the size of the program
- To mitigate the problem we employ *learning*



We use information from already traversed (symbolic execution) subtree to prune other subtrees

Example: Proving Safety



- Verify $x > 0$ at ⟨6⟩
- Depth-first, left-to-right traversal.
- Underlined constrains not needed to maintain unsatisfiability or satisfaction of property

Interpolation

- HALF Interpolant
  Path-based "weakest precondition"
  (Often easy to compute)
- FULL Interpolant
  Combine half interpolants to become Tree-based
  (Challenge is to obtain compact representation)

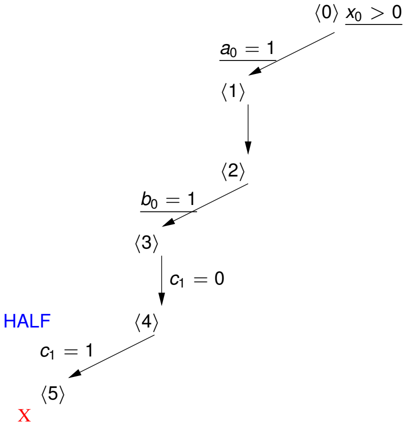
Example of the Most Basic Interpolation Method: UNSAT-CORE

$$x_0 > 0 \ \langle 0 \rangle \ a_0 = 1 \ \langle 1 \rangle \ \langle 2 \rangle \ b_0 = 1 \ \langle 3 \rangle \ c_1 = 0 \ \langle 4 \rangle$$
$$c_1 = 1 \ \langle 5 \rangle \ x_1 = x_0 + 1 \ \langle 6 \rangle$$

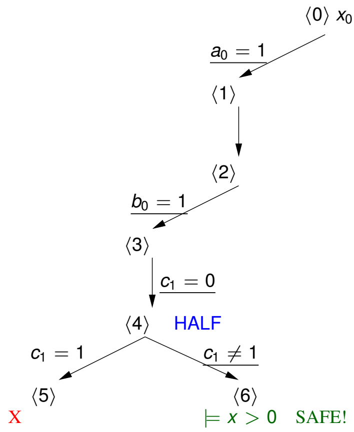
The above constraints are *unsatisfiable*, remove constraints that are not needed to ensure *unsatisfiability*

$$\langle 0 \rangle \ \langle 1 \rangle \ \langle 2 \rangle \ \langle 3 \rangle \ c_1 = 0 \ \langle 4 \rangle \ c_1 = 1 \ \langle 5 \rangle \ \langle 6 \rangle$$

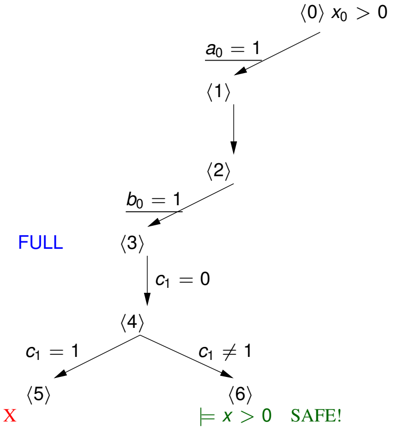
Example: Unsatisfiability Core



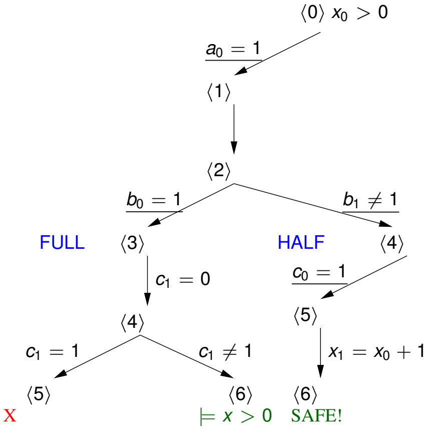
Example: Unsatisfiability Core



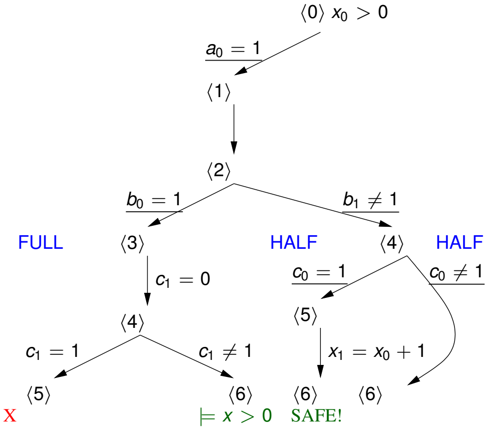
Example: Unsatisfiability Core



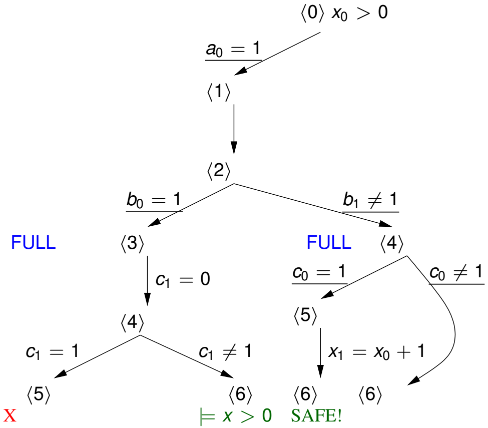
Example: Unsatisfiability Core



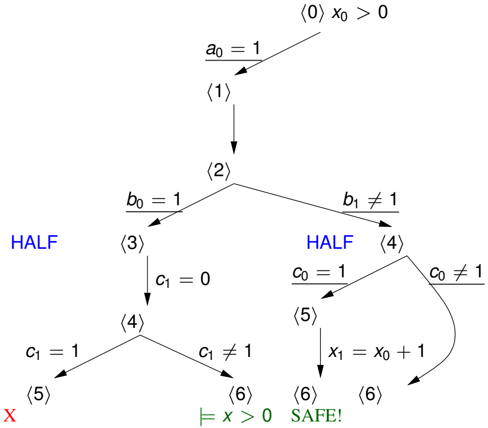
Example: Unsatisfiability Core



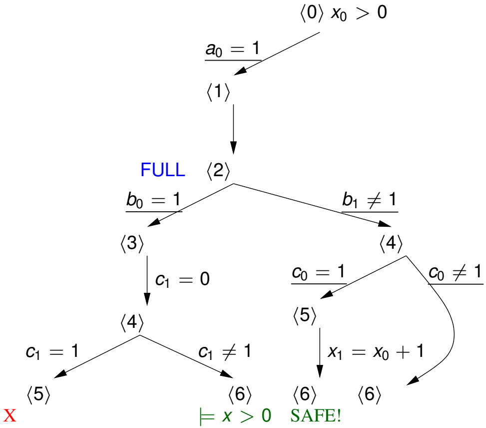
Example: Unsatisfiability Core



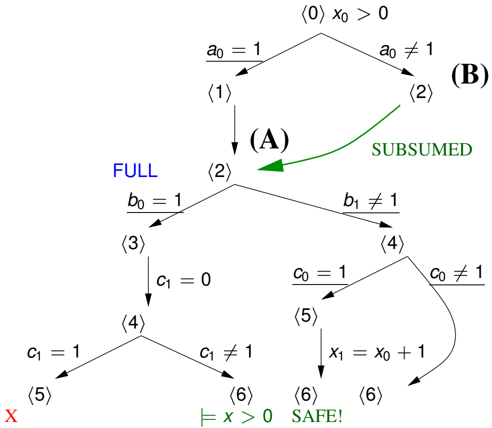
Example: Unsatisfiability Core



Example: Unsatisfiability Core



Example: Unsatisfiability Core



- **W/o interpolation:** (A) is $x > 0, a = 1$, (B) is $x > 0, a \neq 1$, hence (B) not subsumed by (A), big subtree is traversed.
- **W/ interpolation:** (A) is $x > 0$, (B) is $x > 0, a \neq 1$, hence (B) is subsumed by (A), big subtree traversal is avoided.

Interpolation: Weakest Precondition

- Ideal interpolant is the weakest precondition (WP) of the target
- Unfortunately, WP is intractable to compute
- ```
  x = 0;
  if (b1) x += 3 else x += 2
  if (b2) x += 5 else x += 7
  if (b3) x += 9 else x += 14
  {x < 24}
  ```

  Assume $(b1 \wedge \neg b2 \wedge \neg b3)$ is UNSAT.
  WP is:
  $b1 \longrightarrow (\neg b2 \wedge b3 \wedge x \leq 7) \vee (b2 \wedge x \leq 4)$
  $\neg b1 \longrightarrow x < 3$
- Essentially, WP is exponentially disjunctive
  ```

Path-based Weakest Precondition

The general idea: Compute **the weakest precondition of a symbolic state** within the symbolic execution tree by considering only its feasible paths.

suppose a context of $\tilde{c}$ and a postcondition ω:

- $\mathrm{WP}(t, \omega) = \cdots$ inverse transition of t
- $\mathrm{WP}(\mathit{assume}(b), \omega) = \omega \wedge b$
- $\mathrm{WP}(\texttt{if (b) then S1 else S2, } \omega) = \omega \wedge b$ where $\tilde{c} \models b$
- Similarly for when $\tilde{c} \models \neg b$

Conjunctive Path-based Weakest Precondition

The General Case:

`if (b) then S1 else S2` with postcondition ω where

- the context is $\tilde{c} = c_1, c_2, \cdots, c_n$.
- Neither $c \models b$ nor $c \models \neg b$ holds.
- $wpp(S1, \omega)$ is ω_1 and $wpp(S2, \omega)$ is ω_2

In general, the weakest precondition Ψ is a disjunction:

$$(b \longrightarrow \omega_1) \wedge (\neg b \longrightarrow \omega_2)$$

We want to compute a convex Φ. (Therefore $\tilde{c} \models \Phi \models \Psi$)

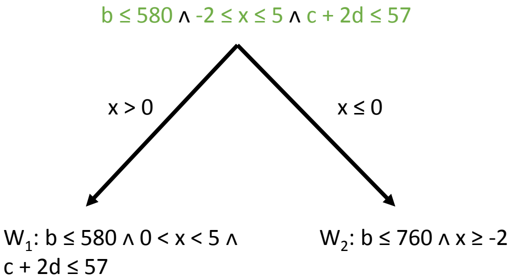
Takeaway:

- There is no succinct definition for this convex.
- The above examples show, however, that there are many special cases to exploit.

Interpolation Example

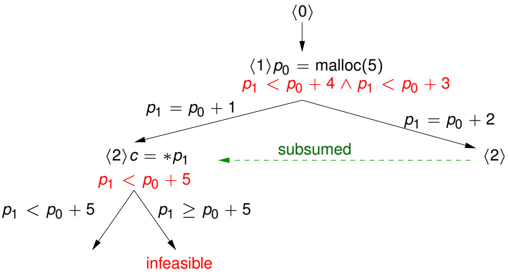
1. Choose a candidate to generalize:
   $c = 2 \wedge d = 4$
2. Extract the subset of W_1 and W_2 which share the same variables with $c = 2 \wedge d = 4$:
   - Subset of W_1: $c + 2d \leq 57$
   - Subset of W_2: $\{\}$
3. If one subset is empty, generalize the candidate to the other subset:
   $c + 2d \geq 57$.

Original Context:
$a > 0 \wedge b = 5 \wedge -1 \leq x \leq 1 \wedge c = 2 \wedge d = 4$



Memory Bounds Interpolation

$\langle 0 \rangle \quad p = \texttt{malloc}(5)$
$\langle 1 \rangle \quad$ **if** $(\ldots)$ **then**
$\quad\quad p{+}{+}$
$\quad$ **else**
$\quad\quad p{+}= 2$
$\quad$ **endif**
$\langle 2 \rangle \quad c := *p$

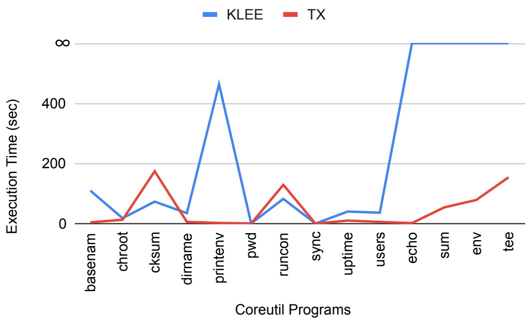


From KLEE TO TRACER-X

- Forward Symbolic Execution to find feasible paths (Similar to KLEE)
- Half interpolants are generated by backward tracking
- Half interpolants merged to full interpolants and stored in Subsumption Table
- Full interpolants used for subsumption at similar program points
- Intermediate execution states should be preserved (Unlike KLEE)

COREUTILS Results 1 (Complete Runs)

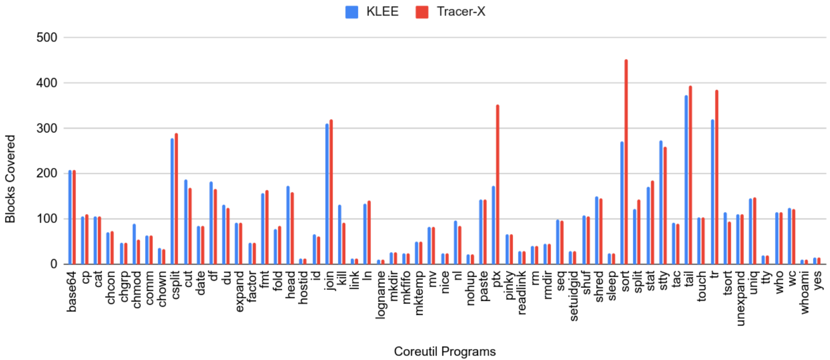
Figure: TRACER-X Finishes Execution but KLEE might not Finish



- **For 12 out of 14 programs** Tracer-X is faster as compared to KLEE.
- Both methods have same coverage, we compare on Analysis Time.

COREUTILS Results 2 (INCOMPLETE Runs)

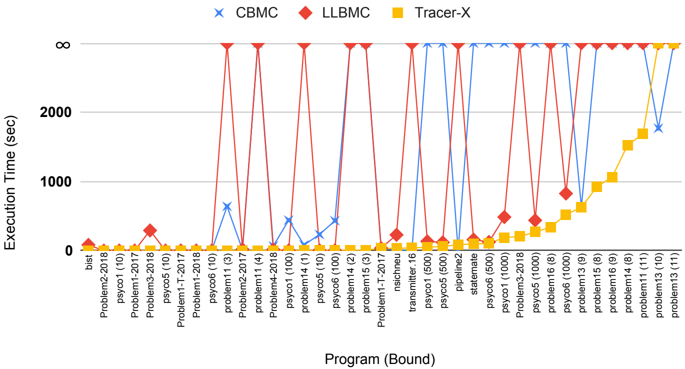
Figure: Both TRACER-X and KLEE do not Finish (62 programs)



- Since both tools timeout, we compare on block coverage
- Tracer-X has higher block coverage **in 16 programs**
- KLEE has higher block coverage **in 14 programs**
- Tracer-X **finds 3 new errors** that are not found by KLEE: kill.c (Line 176), hostid (Line 362, socketcalls.c) and sort (Line 24, memmove.c).

Comparison on SV-COMP

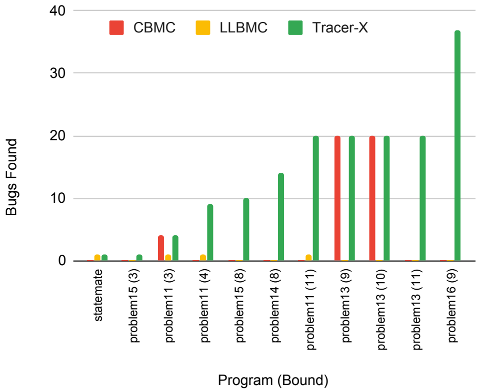
Figure: Comparison with LLBMC and CBMC: Analysis time (20 programs and 39 runs)



- Tracer-X **terminates in 37** while CBMC / LLBMC in 20/23.
- Tracer-X is **faster in 24** while slower in 3/1 (Where terminating).
- Aggregating the terminating cases, Tracer-X was about **3.1X faster** than CBMC and **2.15X faster** than LLBMC.

Comparison on SV-COMP

Figure: Comparison with LLBMC and CBMC: Bug finding (6 programs and 11 runs)



- Tracer-X finds 109 more bugs than CBMC/LLBMC.

Conclusion

- Mitigating state-space blowup using interpolation

- **Implementation:** TRACER-X with Unsat-Core & Conjunctive Weakest Precondition interpolation
  `https://github.com/tracer-x/klee`

- **Applications:**
- **Optimal MC/DC Test Case Generations** (ICSE 2019)
- **A Progressive Quantitative Analysis:** Used to ensure non-functional properties of embedded systems (Submitted to PLDI 2020)
- **Speculative Abstraction for Symbolic Execution** (Submitted to PLDI 2020)
- **Test Generation for Path-Sensitive Code Coverage:** Generating a minimal set of test-cases needed for Modified Condition/Decision Coverage (MC/DC) testing criterion (Submitted to ISSTA 2020)