Enhancing Symbolic Execution with Machine-Checked Safety Proofs

---

David Trabish Shachar Itzhaky

Technion, Israel

CPP 2026

Symbolic Execution

Program analysis technique

- Systematically explores paths
- Checks feasibility using SMT

Applications

- Test input generation
- Bug finding
- Verification

Symbolic Execution

```
int x; // symbolic
int y = 0;
if (x > 7) {
  y++;
  if (x < 1) {
    // ...
  }
}
```

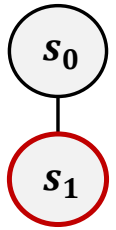
Symbolic Execution

```
int x; // symbolic
int y = 0;
if (x > 7) {
  y++;
  if (x < 1) {
    // ...
  }
}
```

s_0

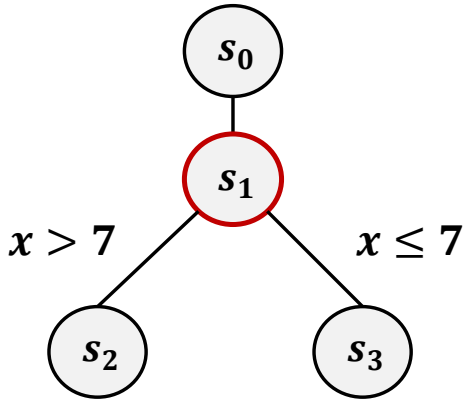
Symbolic Execution

```
int x; // symbolic
int y = 0;
if (x > 7) {
  y++;
  if (x < 1) {
    // ...
  }
}
```



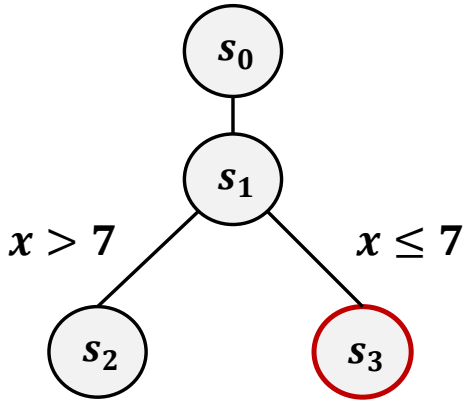
Symbolic Execution

```
int x; // symbolic
int y = 0;
if (x > 7) {
  y++;
  if (x < 1) {
    // ...
  }
}
```



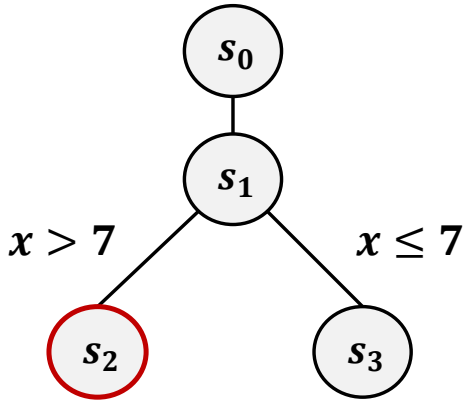
Symbolic Execution

```
int x; // symbolic
int y = 0;
if (x > 7) {
  y++;
  if (x < 1) {
    // ...
  }
}
```



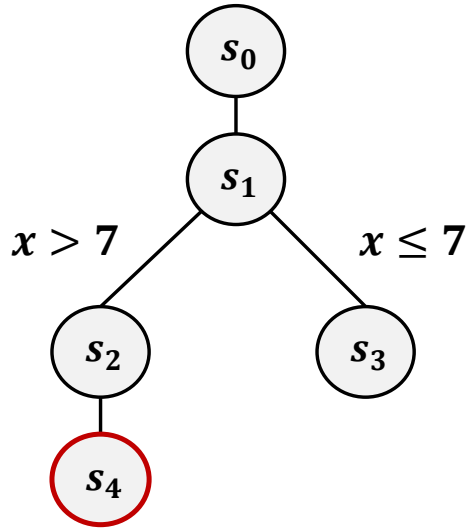
Symbolic Execution

```
int x; // symbolic
int y = 0;
if (x > 7) {
  y++;
  if (x < 1) {
    // ...
  }
}
```



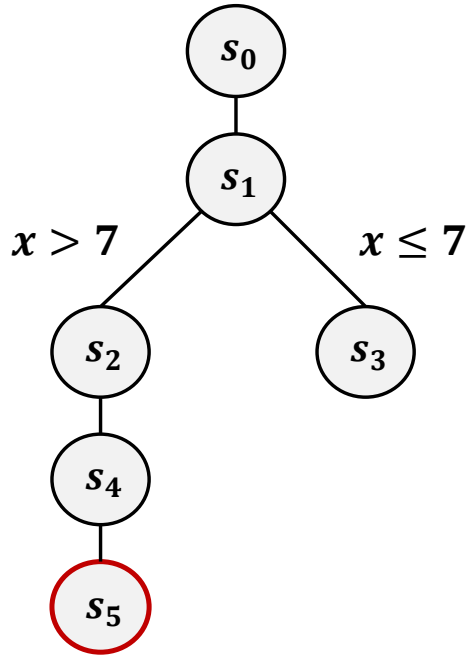
Symbolic Execution

```
int x; // symbolic
int y = 0;
if (x > 7) {
  y++;
  if (x < 1) {
    // ...
  }
}
```



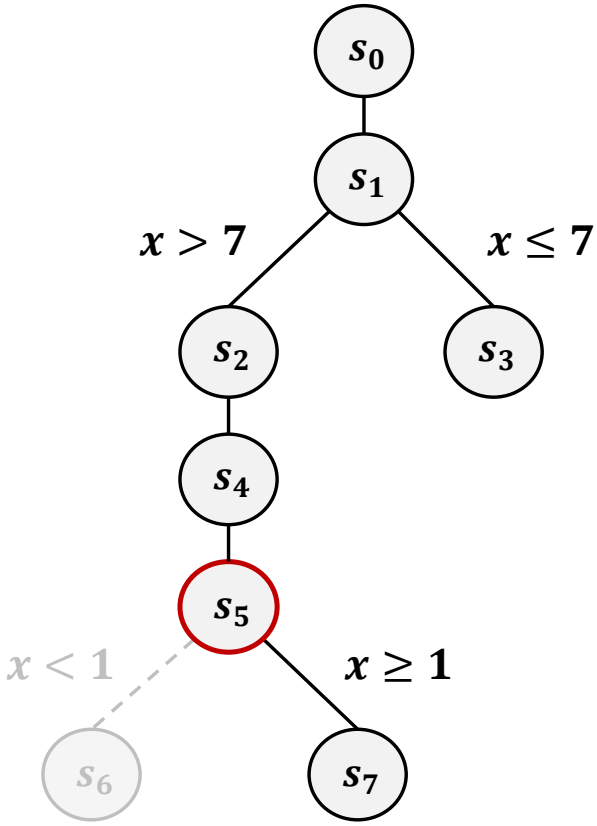
Symbolic Execution

```
int x; // symbolic
int y = 0;
if (x > 7) {
  y++;
  if (x < 1) {
    // ...
  }
}
```



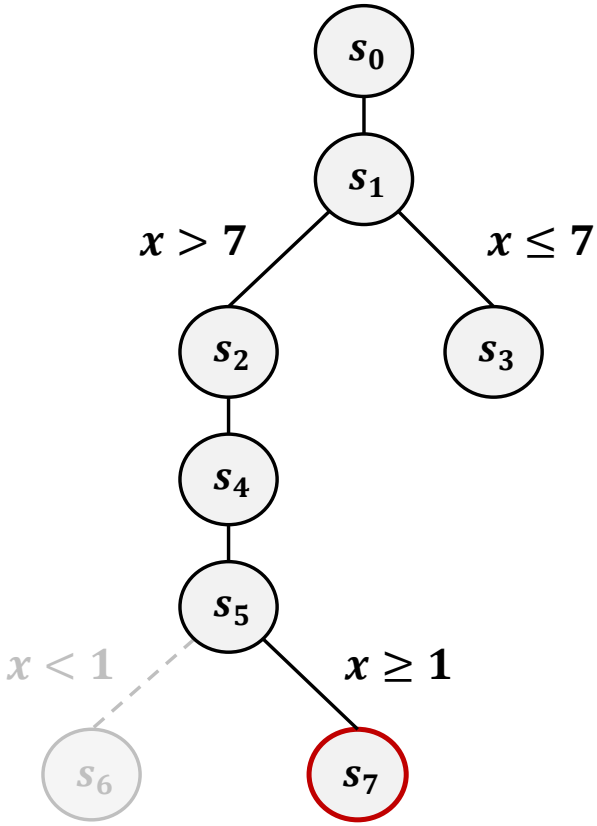
Symbolic Execution

```
int x; // symbolic
int y = 0;
if (x > 7) {
  y++;
  if (x < 1) {
    // ...
  }
}
```



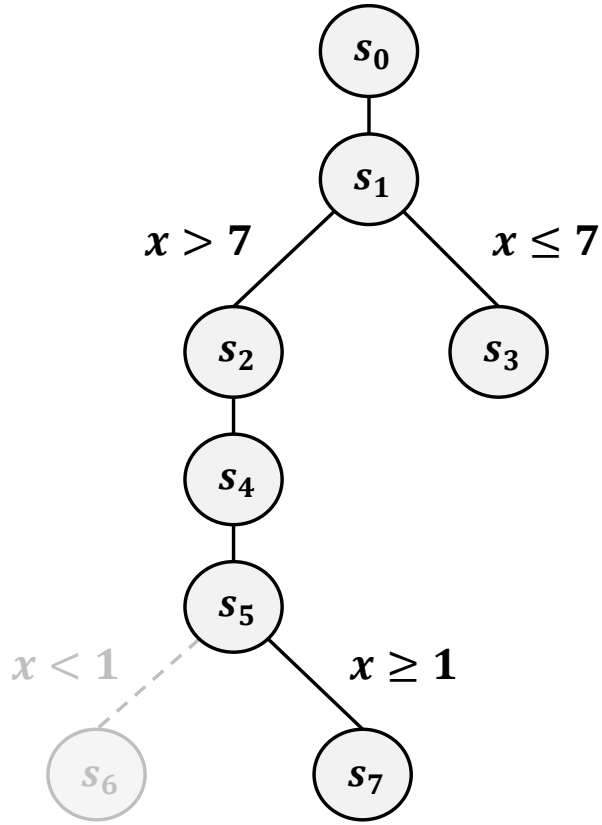
Symbolic Execution

```
int x; // symbolic
int y = 0;
if (x > 7) {
  y++;
  if (x < 1) {
    // ...
  }
}
```

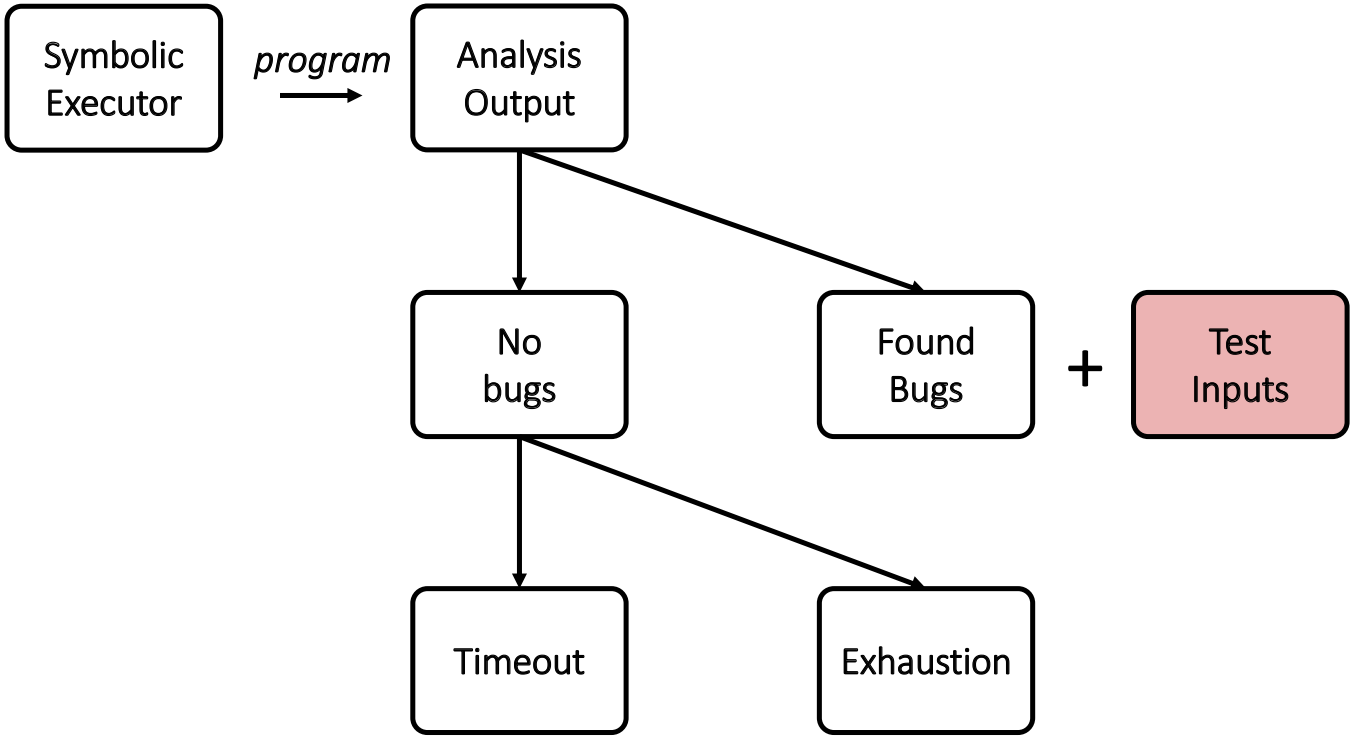


Symbolic Execution

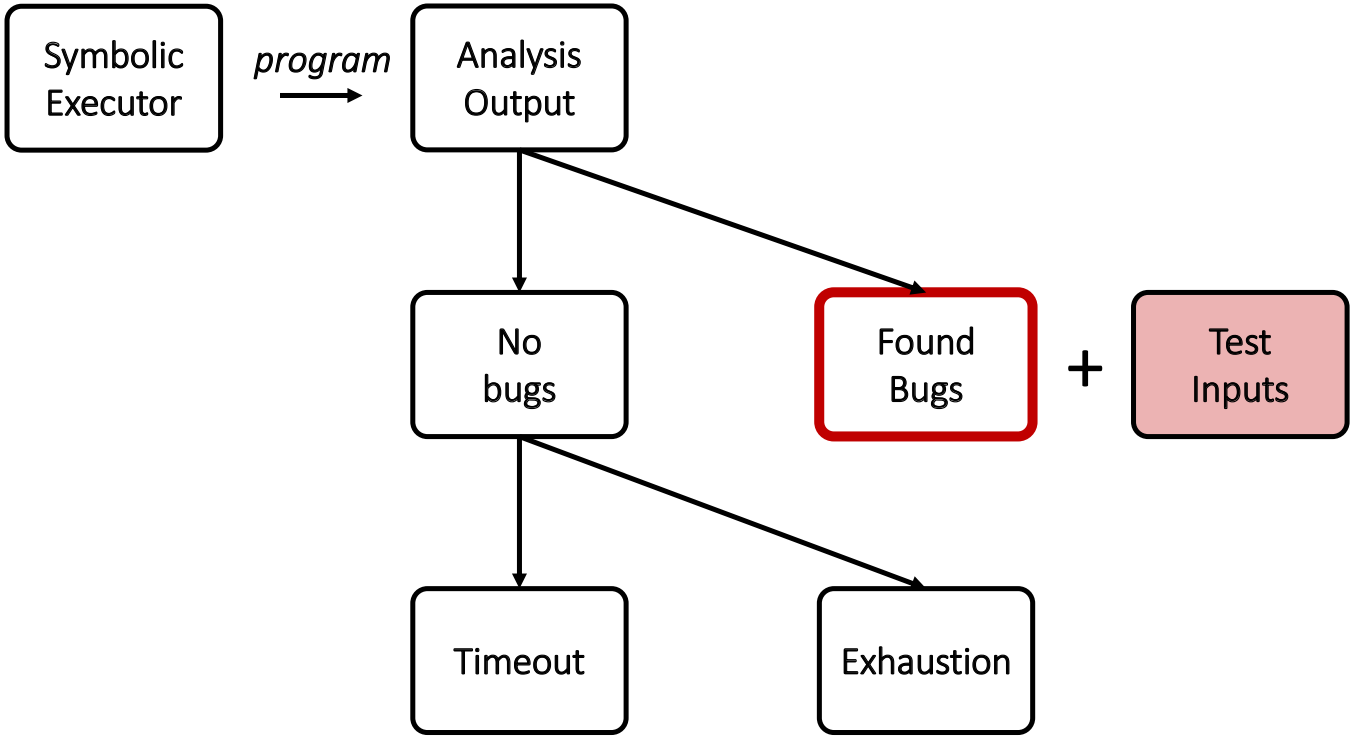
```
int x; // symbolic
int y = 0;
if (x > 7) {
  y++;
  if (x < 1) {
    // ...
  }
}
```



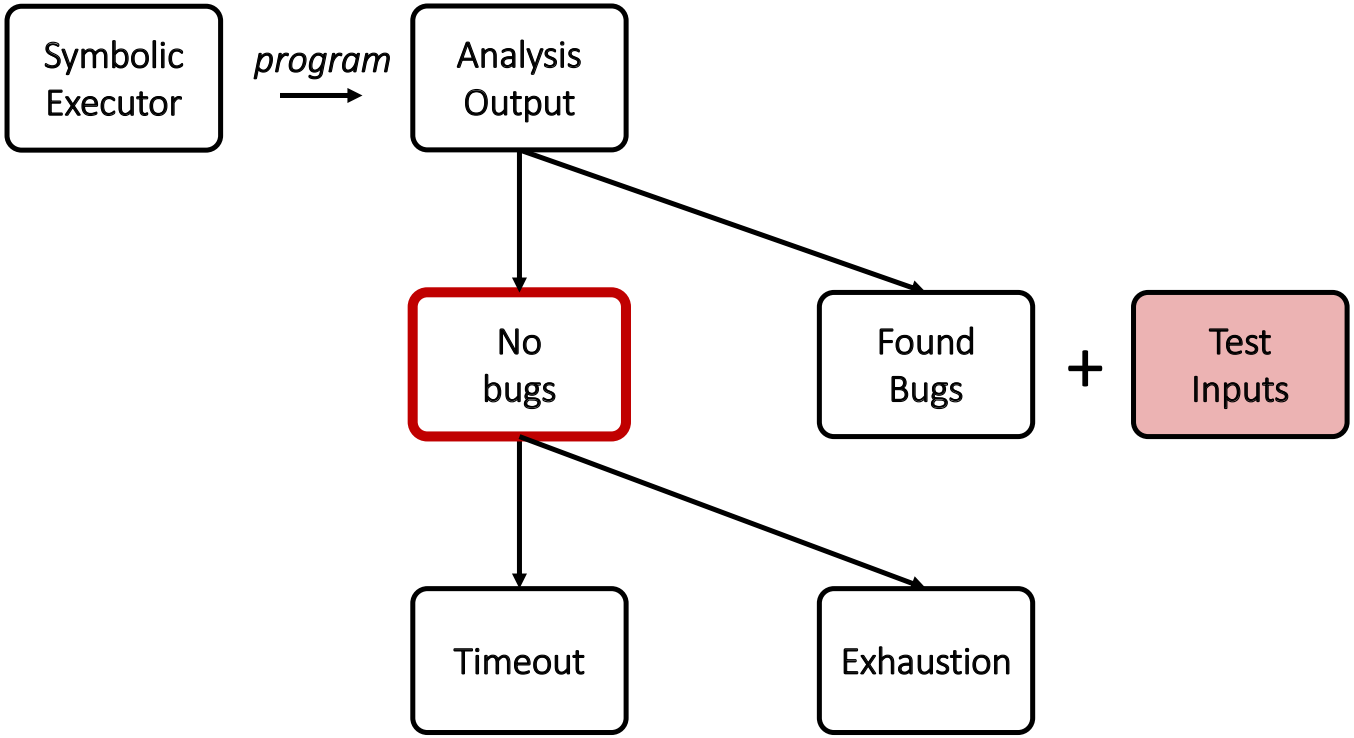
Symbolic Execution



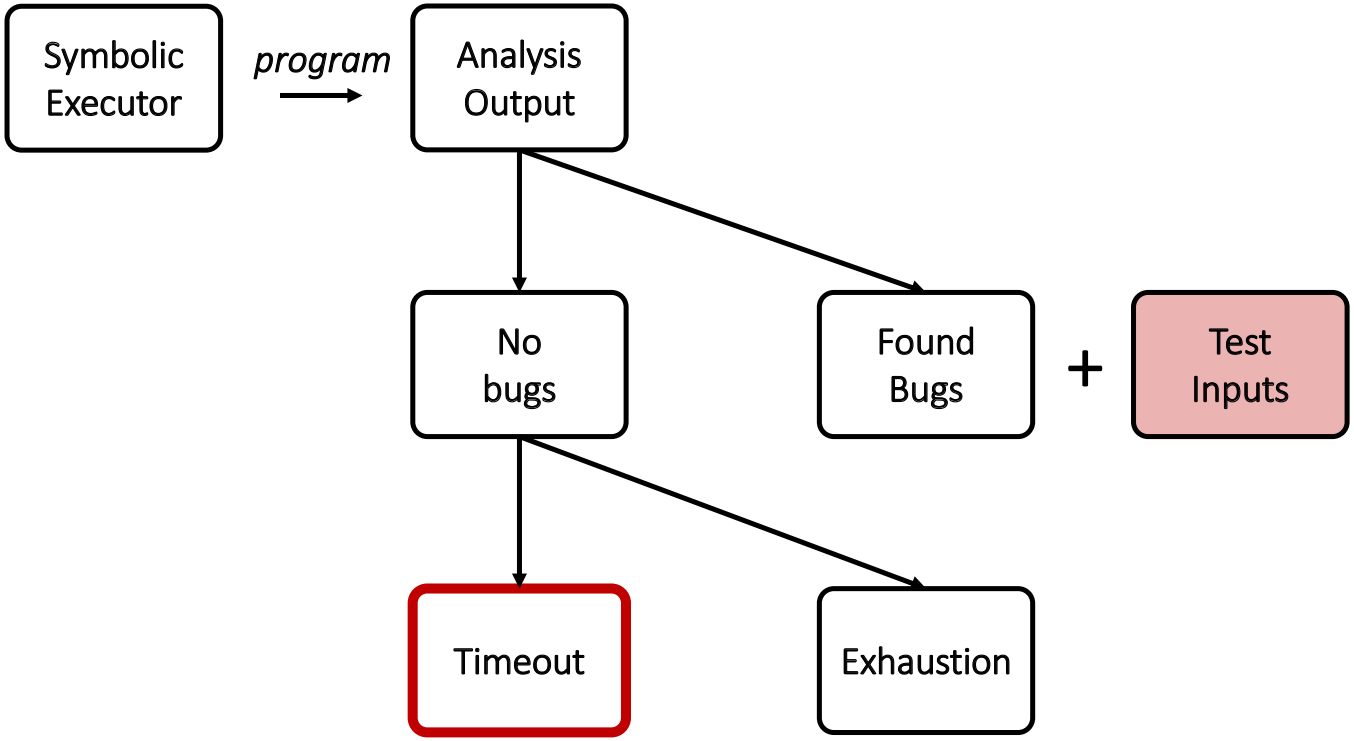
Symbolic Execution



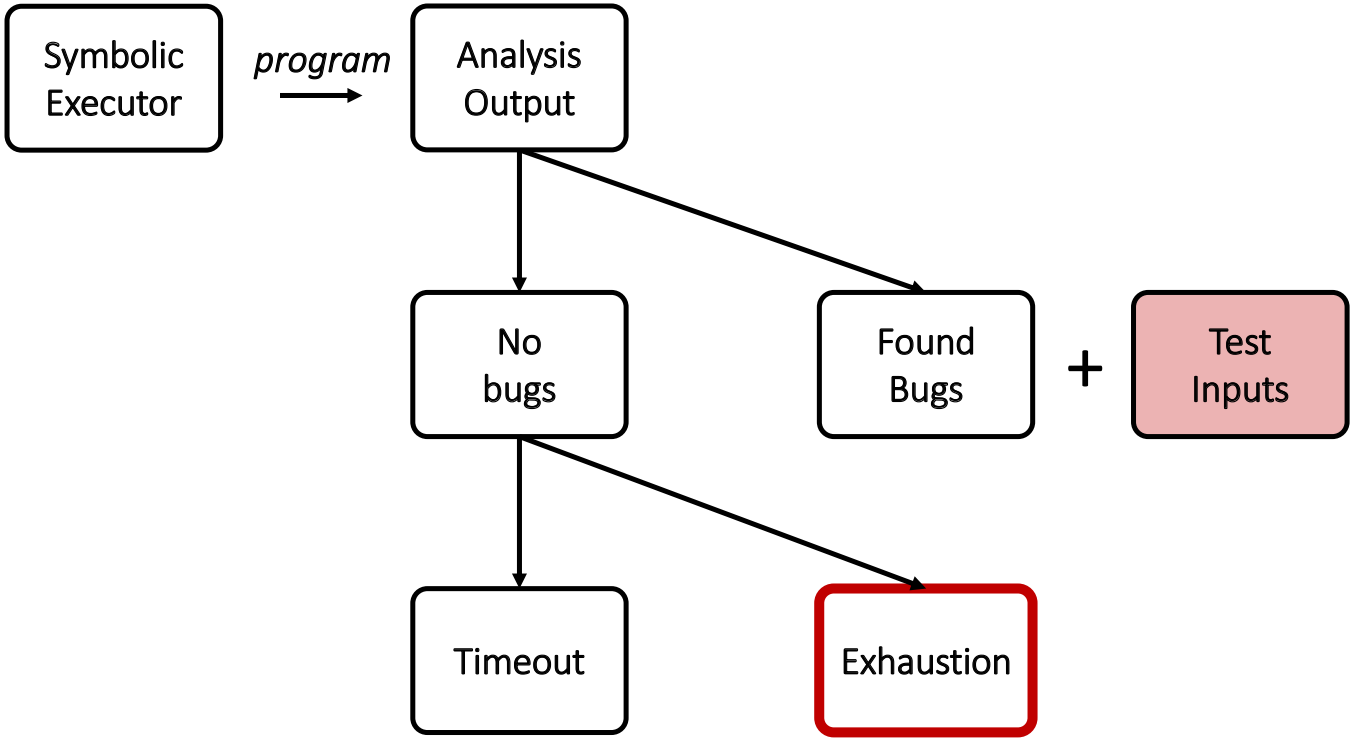
Symbolic Execution



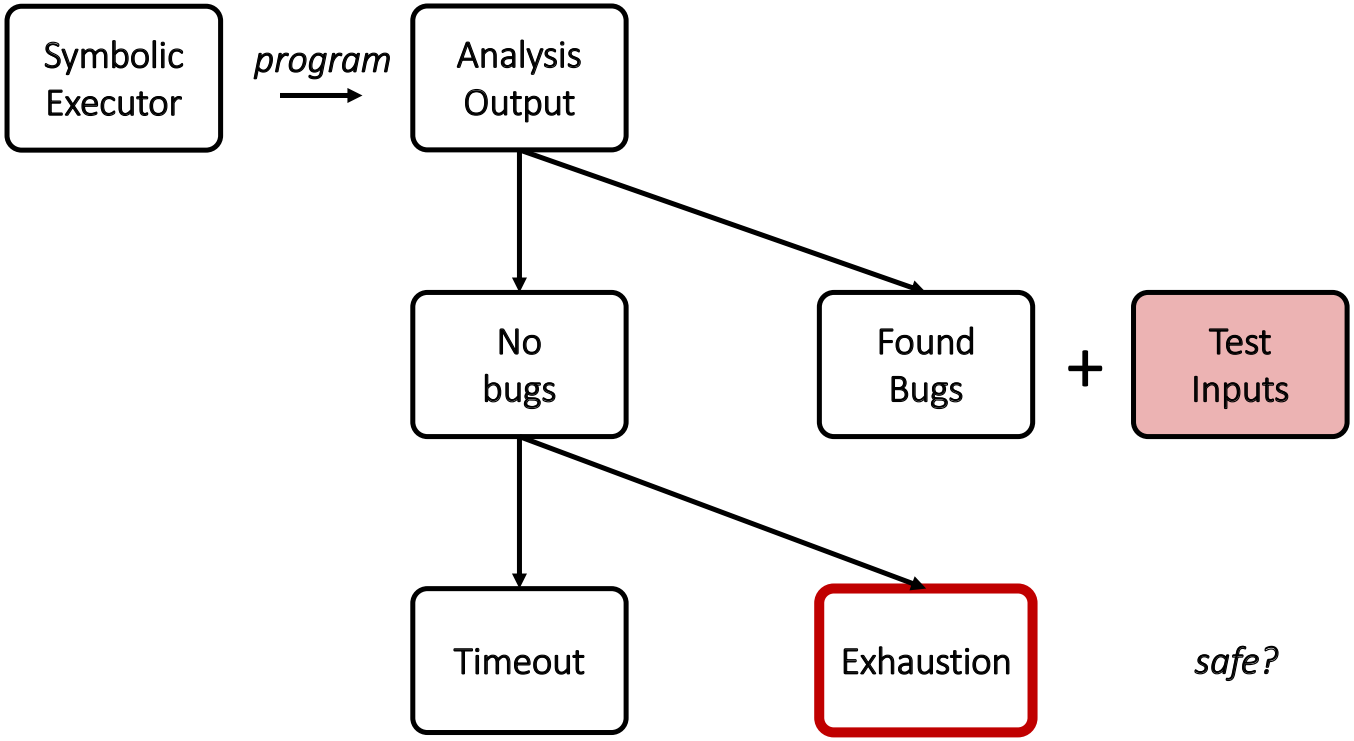
Symbolic Execution



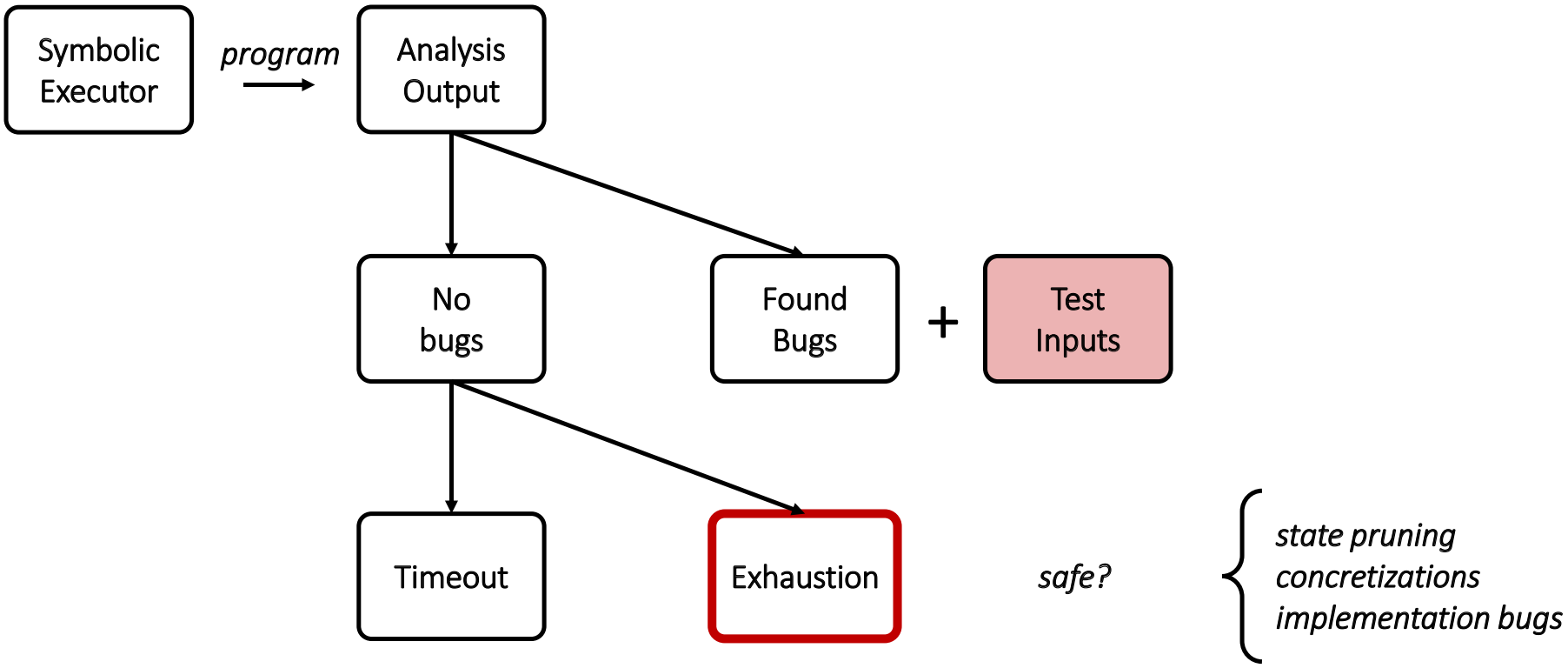
Symbolic Execution



Symbolic Execution



Symbolic Execution



Motivation

```
int x1, x2, y1, y2; // symbolic
if (x1 != x2) {
  int gradient = (y1 – y2) / (x1 – x2);
}
```

Motivation

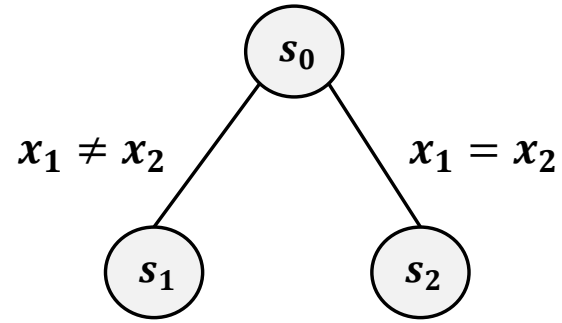
```
int x1, x2, y1, y2; // symbolic
if (x1 != x2) {
  int gradient = (y1 - y2) / (x1 - x2);
}
```

s_0

computed by KLEE

Motivation

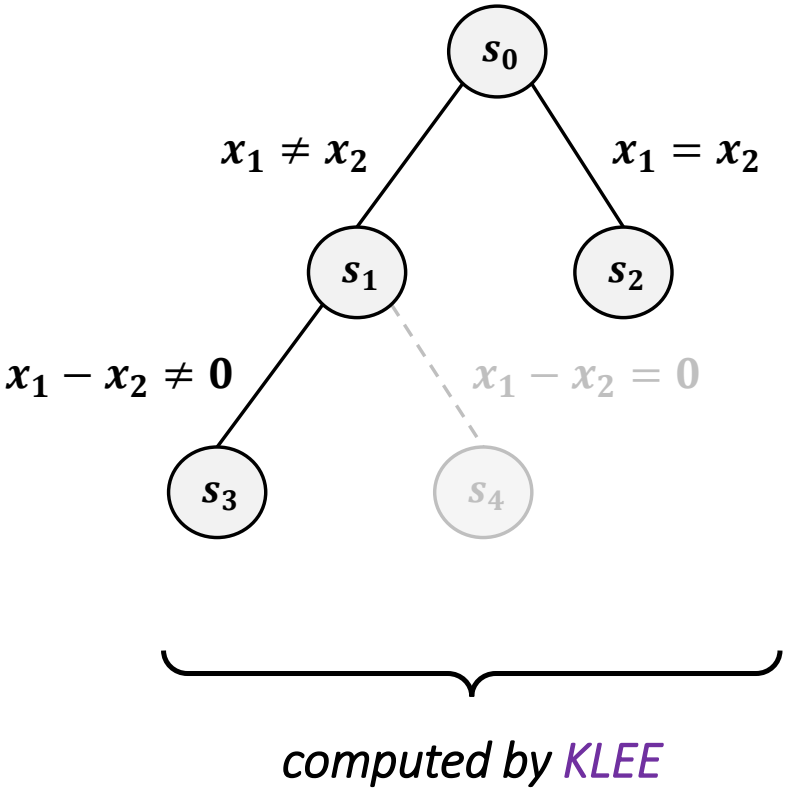
```
int x1, x2, y1, y2; // symbolic
if (x1 != x2) {
  int gradient = (y1 - y2) / (x1 - x2);
}
```



computed by KLEE

Motivation

```
int x1, x2, y1, y2; // symbolic
if (x1 != x2) {
  int gradient = (y1 - y2) / (x1 - x2);
}
```

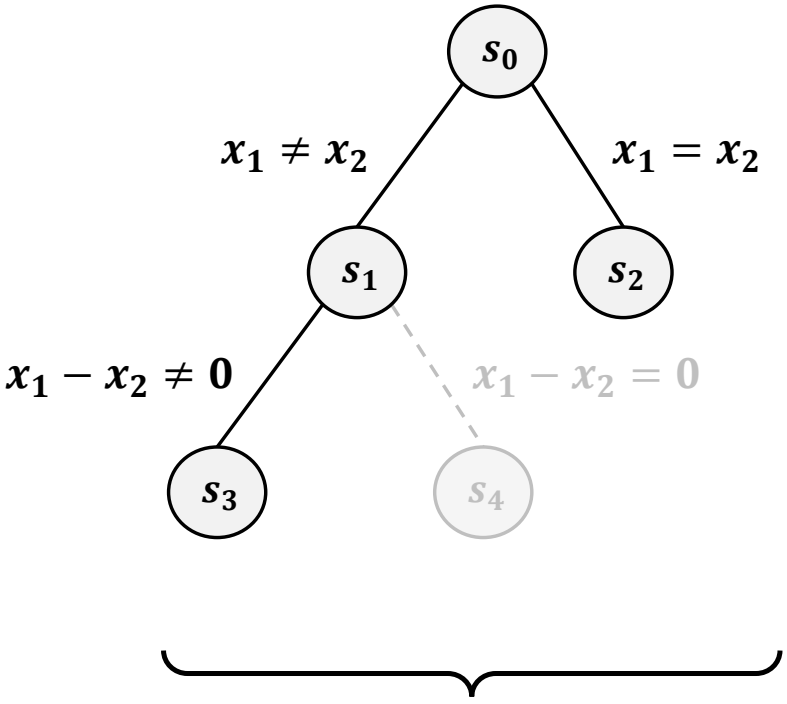


computed by KLEE

Motivation

```
int x1, x2, y1, y2; // symbolic
if (x1 != x2) {
  int gradient = (y1 - y2) / (x1 - x2);
}
```

UNSAFE



computed by KLEE

signed division a/b fails when:

☑ $b = 0$

☒ $a = -2147483647,\ b = -1$

How can we guarantee safety?

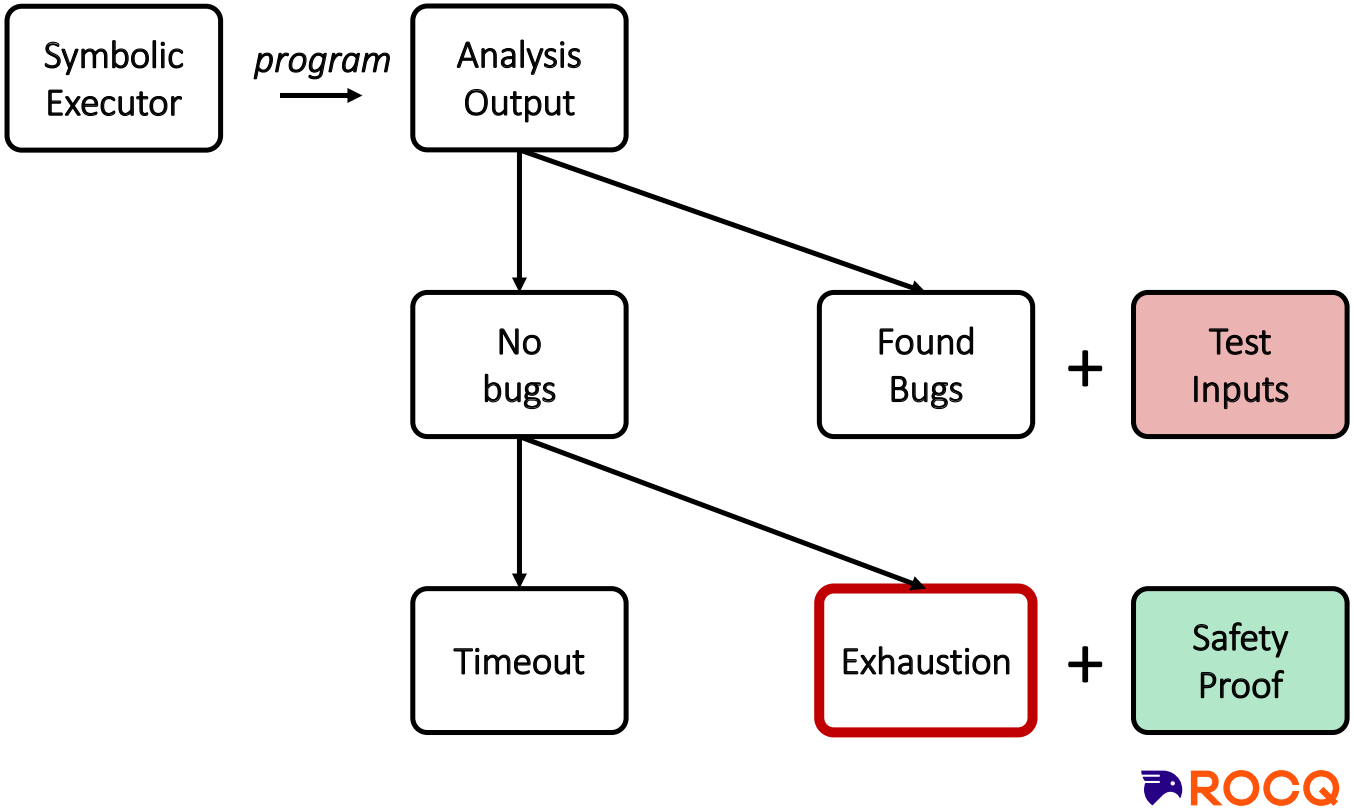
Verifying the symbolic executor is hard

- Complex code with third-party dependencies
  - *Standard library, SMT Solver, etc.*

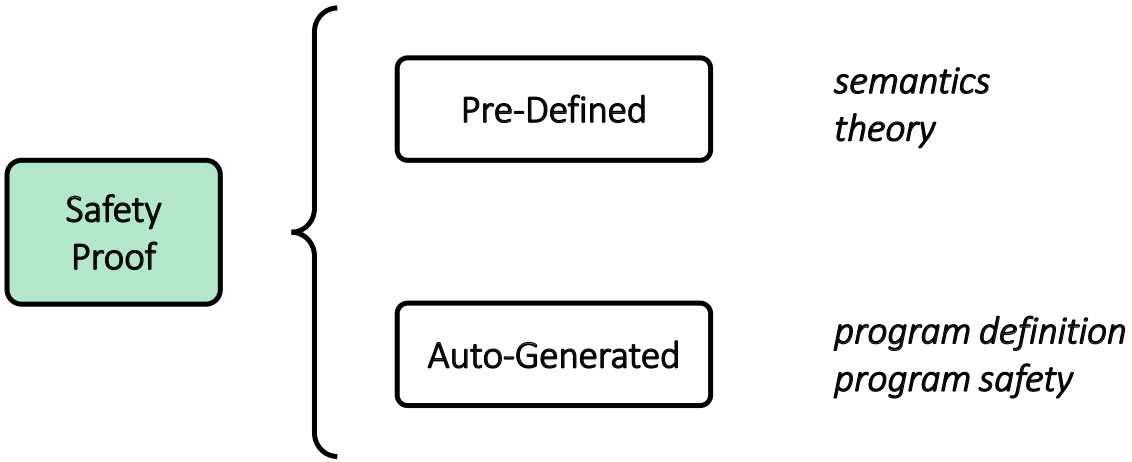
Our Solution:

- Adopt the approach of ***translation validation***
- Generate a safety proof that is specific to the analyzed program

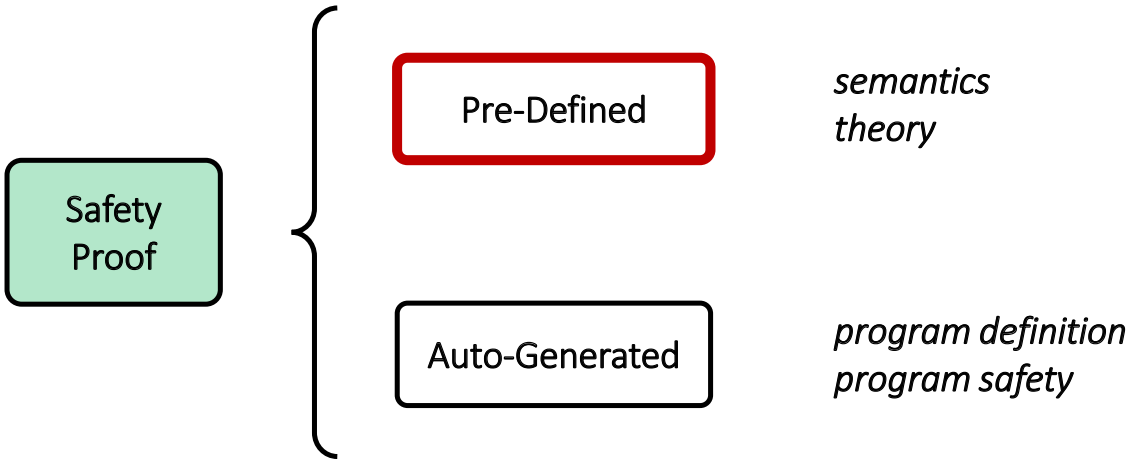
Our Approach



Proof Structure



Proof Structure



Concrete and Symbolic Semantics

- Concrete state: (ℓ, σ, κ)
  - ℓ: program location
  - σ: store (integers, undef, poison)
  - κ: stack

- Symbolic state: $(\ell, \tilde{\sigma}, \tilde{\kappa}, \varphi)$
  - ℓ: program location
  - $\tilde{\sigma}$: symbolic store (SMT expressions)
  - $\tilde{\kappa}$: symbolic stack
  - φ: path constraints

Concrete

Assign Rule $(\ell : x = e)$

$$\frac{}{(\ell, \sigma, \kappa) \to (\ell + 1, \sigma[x \mapsto eval(e, \sigma)], \kappa)}$$

Branch Rule $(\ell : br\ e\ b_1\ b_2)$

$$\frac{eval(e, \sigma) = true}{(\ell, \sigma, \kappa) \to (\ell_{b_1}, \sigma, \kappa)}$$

$$\frac{eval(e, \sigma) = false}{(\ell, \sigma, \kappa) \to (\ell_{b_2}, \sigma, \kappa)}$$

Symbolic

$$\frac{}{(\ell, \tilde{\sigma}, \tilde{\kappa}, \varphi) \rightsquigarrow (\ell + 1, \tilde{\sigma}[x \mapsto \widetilde{eval}(e, \tilde{\sigma})], \tilde{\kappa}, \varphi)}$$

$$\frac{}{(\ell, \tilde{\sigma}, \tilde{\kappa}, \varphi) \rightsquigarrow (\ell_{b_1}, \tilde{\sigma}, \tilde{\kappa}, \varphi \wedge \widetilde{eval}(e, \tilde{\sigma}))}$$

$$\frac{}{(\ell, \tilde{\sigma}, \tilde{\kappa}, \varphi) \rightsquigarrow (\ell_{b_2}, \tilde{\sigma}, \tilde{\kappa}, \varphi \wedge \neg\widetilde{eval}(e, \tilde{\sigma}))}$$

Concrete and Symbolic Semantics

Over-approximation relation $(\ell, \sigma, \kappa) \succ (\ell, \tilde{\sigma}, \tilde{\kappa}, \varphi)$:

- $m \vDash \varphi$
- c is a concretization of s using m

Theorem (Reachability Completeness):

$$c_{init} \rightarrow^* c$$

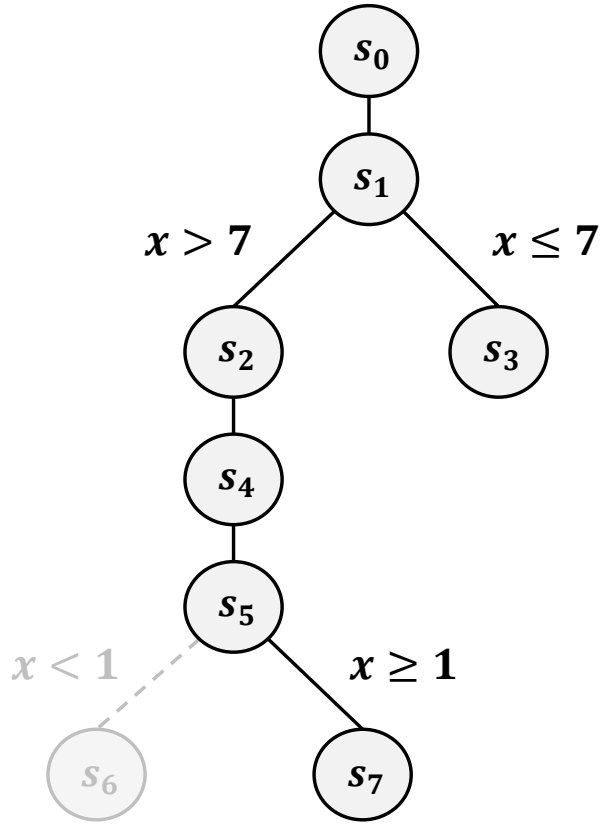
$$(and\ there\ are\ no\ poisoned\ states)$$

$$\Downarrow$$

$$\exists s.\ (s_{init} \tilde{\rightarrow}^* s) \wedge (s \succ c)$$

Execution Trees

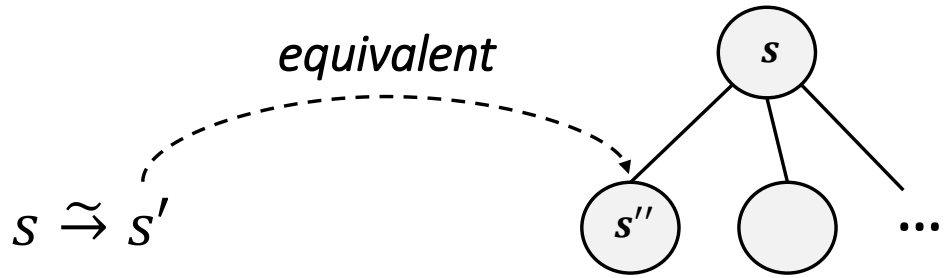
```
int x; // symbolic
int y = 0;
if (x > 7) {
  y++;
  if (x < 1) {
    // ...
  }
}
```



Safe Execution Trees

The subtree of s is safe if:

- s is not an error state
- If $s \overset{\sim}{\Rightarrow} s'$, one of these must hold:
  - s' is infeasible
  - There a child s'' of s such that:
    - *s' and s'' are equivalent*
    - The subtree of s'' is safe

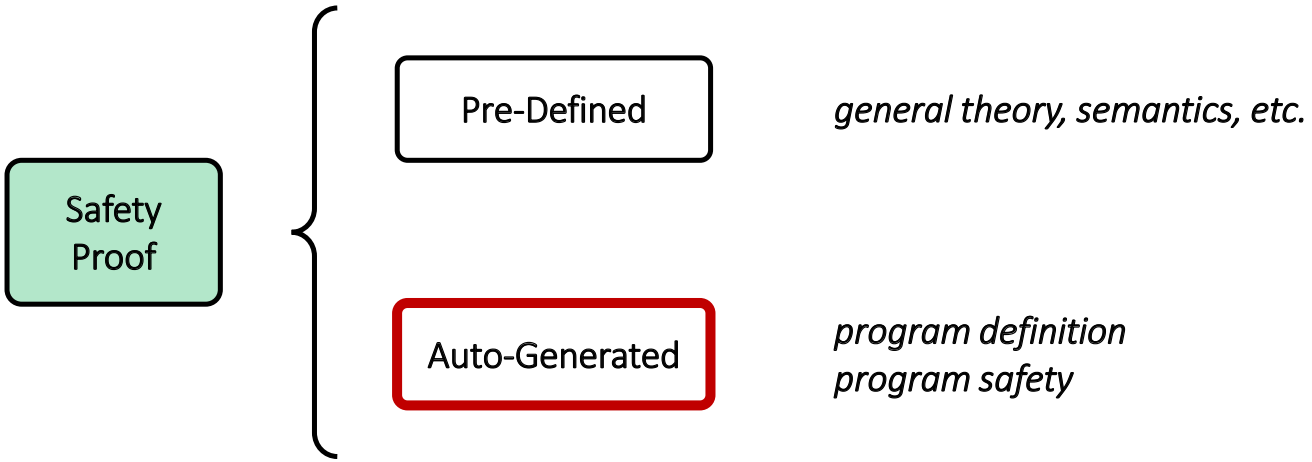


Safe Execution Trees

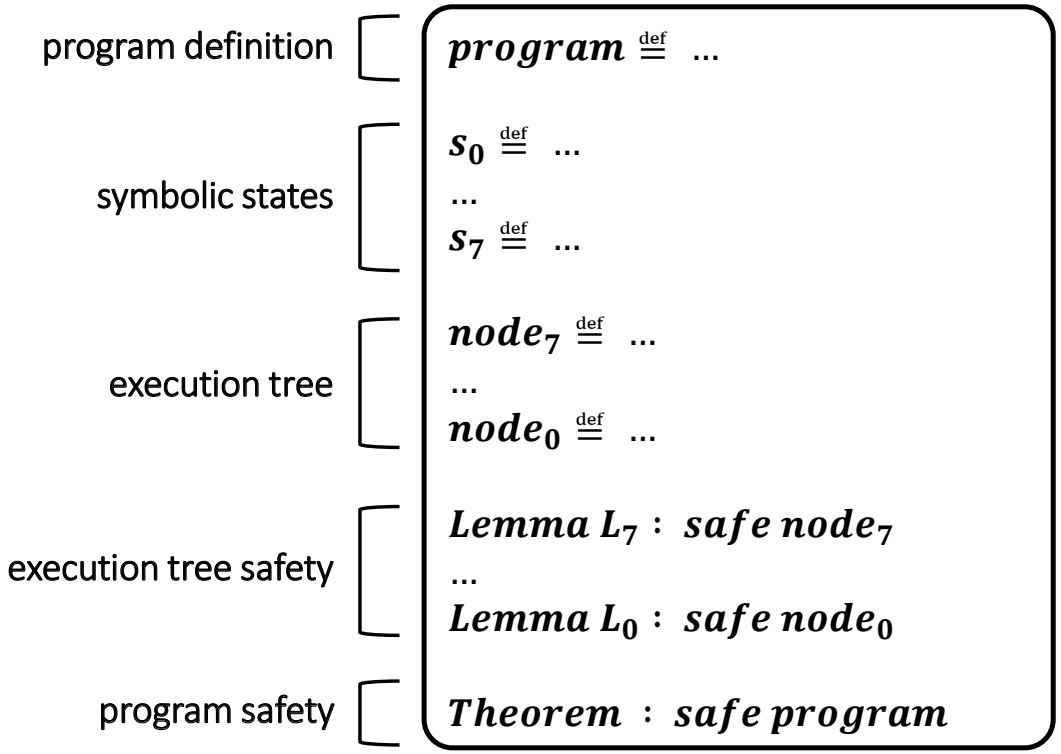
Theorem:

If the execution tree of s_{init} is safe,
then the program is safe w.r.t the concrete semantics.

Proof Structure

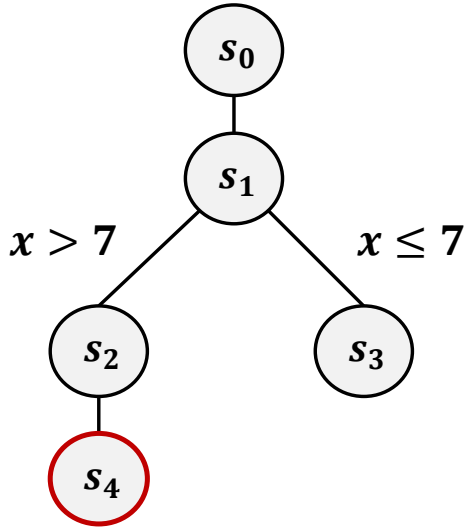


Auto-Generation Proof

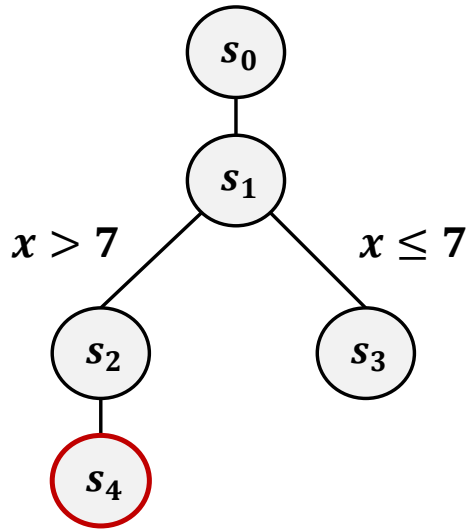


```
int x; // symbolic
int y = 0;
if (x > 7) {
  y++;
  if (x < 1) {
    // ...
  }
}
```

$(\ell_4, [y \mapsto 0], [], x > 0)$



```
int x; // symbolic
int y = 0;
if (x > 7) {
  y++;
  if (x < 1) {
    // ...
  }
}
```

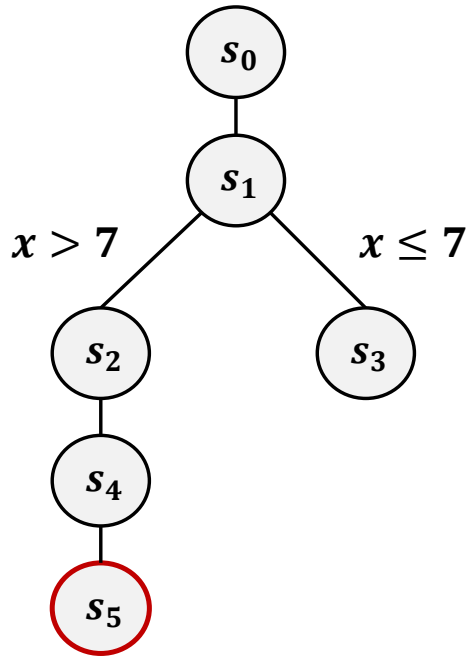


$$\underbrace{(\ell_4, [y \mapsto 0], [], x > 0) \rightsquigarrow (\ell_5, [y \mapsto 0 + 1], [], x > 0)}_{\textit{assign rule}}$$

```
int x; // symbolic
int y = 0;
if (x > 7) {
  y++;
  if (x < 1) {
    // ...
  }
}
```

$$(\ell_4, [y \mapsto 0], [], x > 0) \rightsquigarrow (\ell_5, [y \mapsto 0 + 1], [], x > 0)$$

assign rule

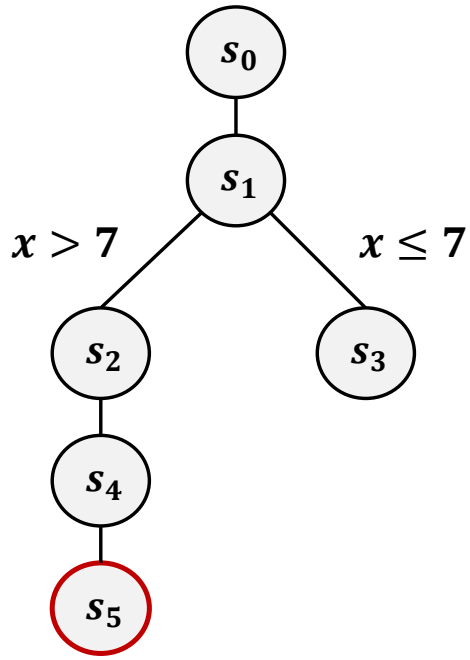


$(\ell_5, [y \mapsto 1], [], x > 0)$

```
int x; // symbolic
int y = 0;
if (x > 7) {
  y++;
  if (x < 1) {
    // ...
  }
}
```

$$(\ell_4, [y \mapsto 0], [], x > 0) \rightsquigarrow (\ell_5, [y \mapsto 0 + 1], [], x > 0)$$

assign rule



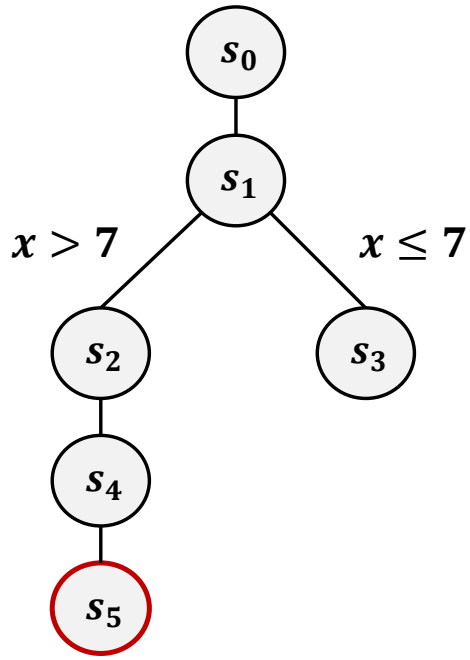
$$(\ell_5, [y \mapsto 1], [], x > 0)$$

```
int x; // symbolic
int y = 0;
if (x > 7) {
  y++;
  if (x < 1) {
    // ...
  }
}
```

$$(\ell_4, [y \mapsto 0], [], x > 0) \rightsquigarrow (\ell_5, [y \mapsto 0 + 1], [], x > 0)$$

assign rule

$$(\ell_5, [y \mapsto 1], [], x > 0)$$

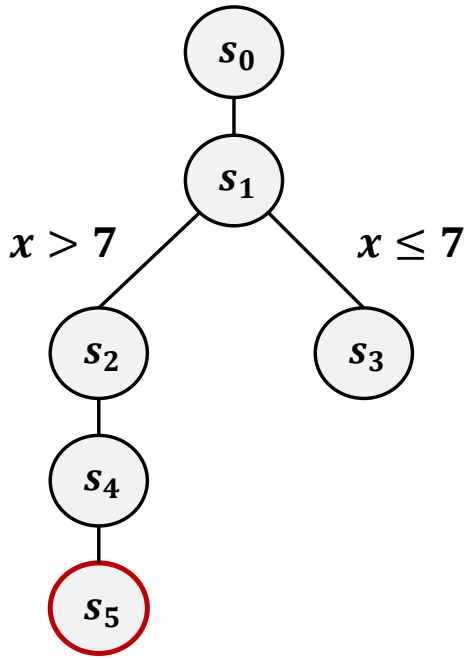


```
int x; // symbolic
int y = 0;
if (x > 7) {
  y++;
  if (x < 1) {
    // ...
  }
}
```

$$(\ell_5, [\ldots], [], x > 7) \rightsquigarrow (\ell_6, [\ldots], [], x > 7 \wedge x < 1)$$

$$(\ell_5, [\ldots], [], x > 7) \rightsquigarrow (\ell_7, [\ldots], [], x > 7 \wedge x \geq 1)$$

branch rule

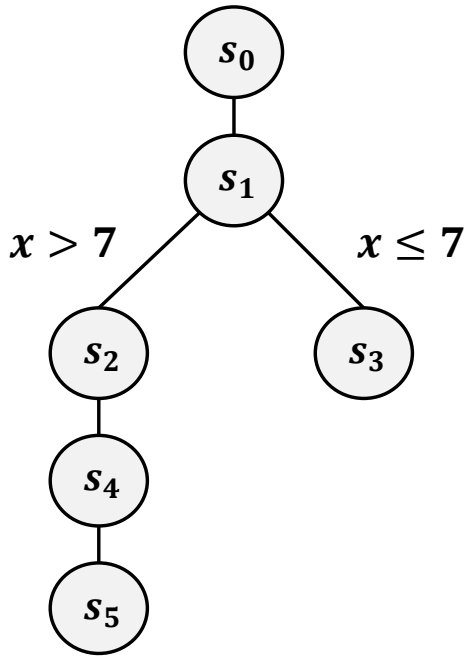


```
int x; // symbolic
int y = 0;
if (x > 7) {
  y++;
  if (x < 1) {
    // ...
  }
}
```

$$(\ell_5, [\ldots], [], x > 7) \rightsquigarrow (\ell_6, [\ldots], [], x > 7 \wedge x < 1)$$

$$(\ell_5, [\ldots], [], x > 7) \rightsquigarrow (\ell_7, [\ldots], [], x > 7 \wedge x \geq 1)$$

branch rule



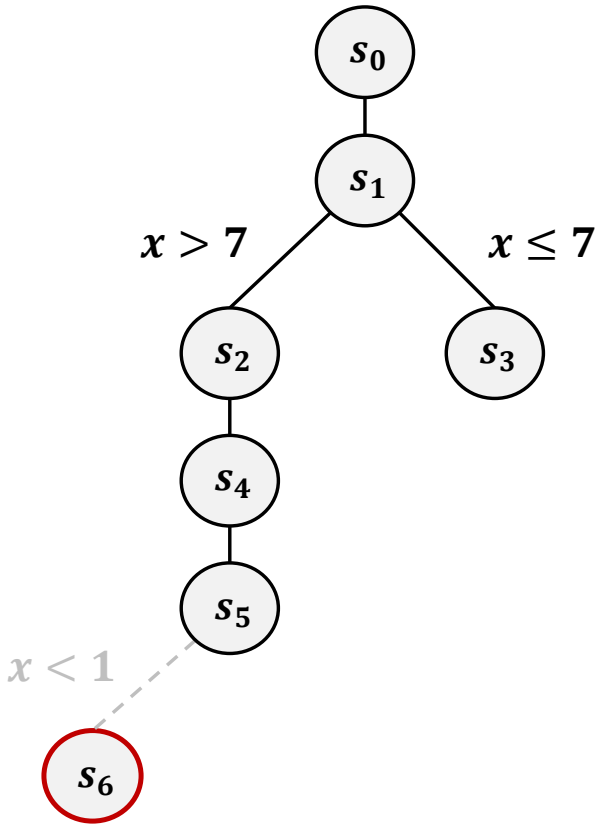
```
int x; // symbolic
int y = 0;
if (x > 7) {
  y++;
  if (x < 1) {
    // ...
  }
}
```

$$(\ell_5, [\ldots], [], x > 7) \rightsquigarrow (\ell_6, [\ldots], [], x > 7 \wedge x < 1)$$

unsat

$$(\ell_5, [\ldots], [], x > 7) \rightsquigarrow (\ell_7, [\ldots], [], x > 7 \wedge x \geq 1)$$

branch rule

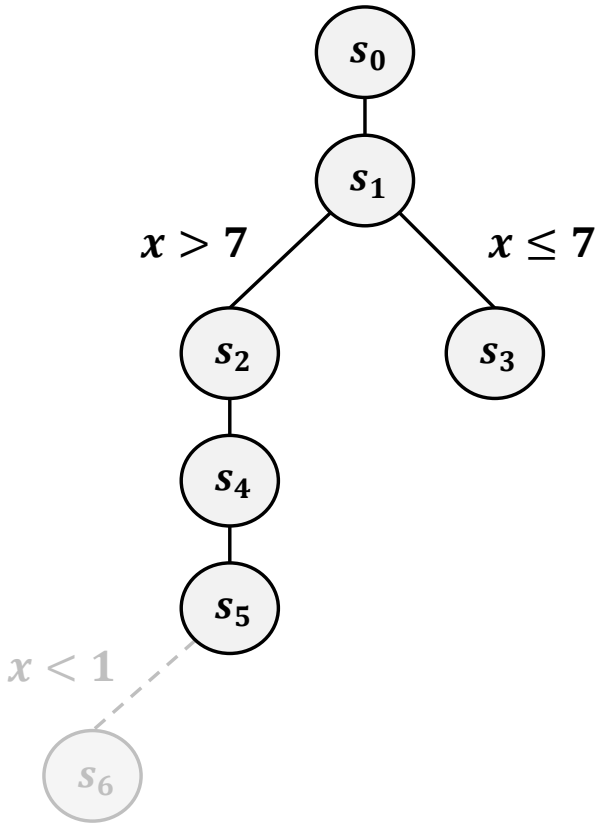


```
int x; // symbolic
int y = 0;
if (x > 7) {
  y++;
  if (x < 1) {
    // ...
  }
}
```

$(\ell_5, [\ldots], [], x > 7) \rightsquigarrow (\ell_6, [\ldots], [], x > 7 \wedge x < 1)$

$(\ell_5, [\ldots], [], x > 7) \rightsquigarrow (\ell_7, [\ldots], [], x > 7 \wedge x \geq 1)$

branch rule



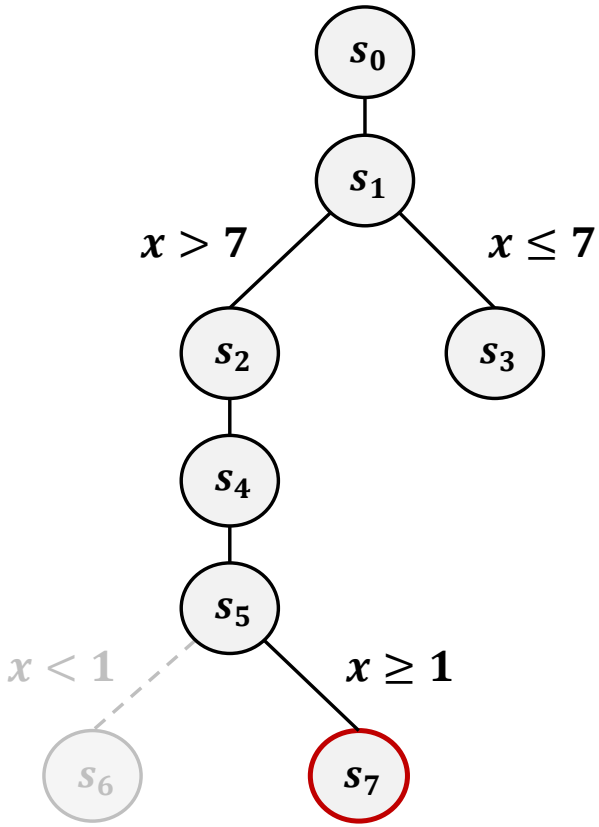
```
int x; // symbolic
int y = 0;
if (x > 7) {
  y++;
  if (x < 1) {
    // ...
  }
}
```

$(\ell_5, [\ldots], [], x > 7) \rightsquigarrow (\ell_6, [\ldots], [], x > 7 \wedge x < 1)$

$(\ell_5, [\ldots], [], x > 7) \rightsquigarrow (\ell_7, [\ldots], [], x > 7 \wedge x \geq 1)$

branch rule

$(\ell_7, [y \mapsto 0], [], x > 7)$



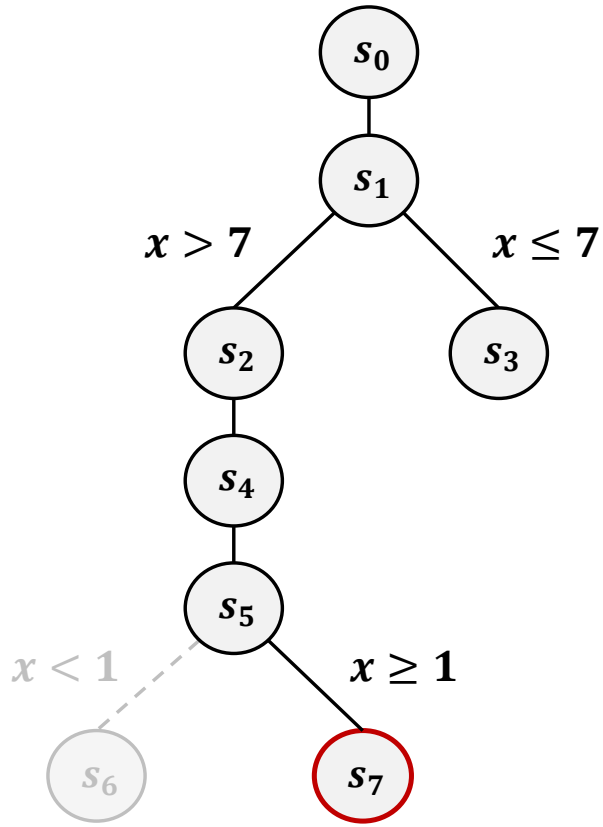
```
int x; // symbolic
int y = 0;
if (x > 7) {
  y++;
  if (x < 1) {
    // ...
  }
}
```

$(\ell_5, [\ldots], [], x > 7) \rightsquigarrow (\ell_6, [\ldots], [], x > 7 \wedge x < 1)$

$(\ell_5, [\ldots], [], x > 7) \rightsquigarrow (\ell_7, [\ldots], [], x > 7 \wedge x \geq 1)$

branch rule

$(\ell_7, [y \mapsto 0], [], x > 7)$

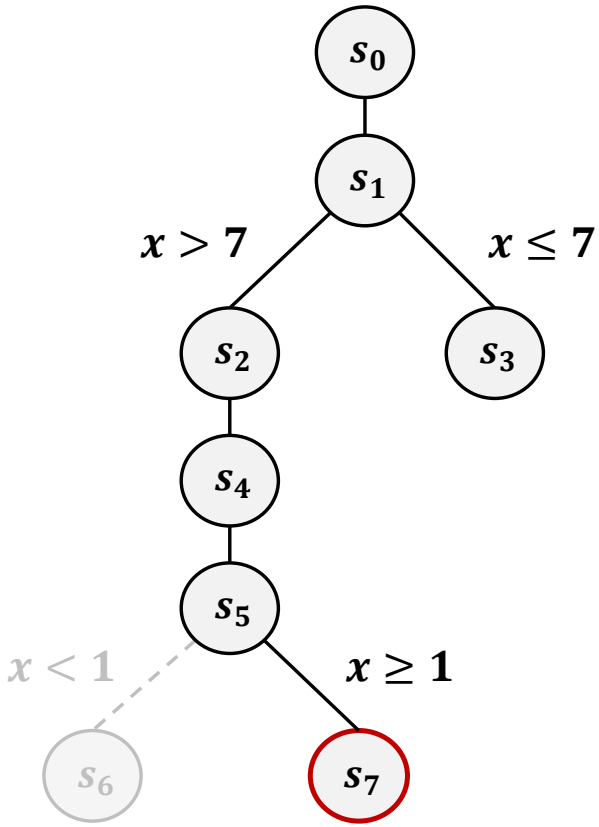


```
int x; // symbolic
int y = 0;
if (x > 7) {
  y++;
  if (x < 1) {
    // ...
  }
}
```

$(\ell_5, [\ldots], [], x > 7) \rightsquigarrow (\ell_6, [\ldots], [], x > 7 \wedge x < 1)$

$(\ell_5, [\ldots], [], x > 7) \rightsquigarrow (\ell_7, [\ldots], [], x > 7 \wedge x \geq 1)$

branch rule



$(\ell_7, [y \mapsto 0], [], x > 7)$

$unsat(pc \wedge \neg e) \Rightarrow pc \wedge e \equiv pc$

Proof Compilation Optimizations

- Term Reuse
- Proof Object Minimization
  - Avoid expensive *Rocq* tactics

$$\underbrace{(\ell_4, [y \mapsto 0], [], x > 0) \rightsquigarrow (\ell_5, [y \mapsto 0 + 1], [], x > 0)}_{\textit{assign rule}}$$

$$(\ell_5, [y \mapsto 1], [], x > 0)$$

```
int x; // symbolic
int y = 0;
if (x > 7) {
  y++;
  if (x < 1) {
    // ...
  }
}
```

$$\underbrace{(\ell_4, [y \mapsto 0], [], x > 0) \rightsquigarrow (\ell_5, [y \mapsto 0 + 1], [], x > 0)}_{\textit{assign rule}}$$

$(\ell_5, [y \mapsto 1], [], x > 0)$

```
int x; // symbolic
int y = 0;
if (x > 7) {
  y++;
  if (x < 1) {
    // ...
  }
}
```

$$\underbrace{(\ell_4, [y \mapsto 0], [], x > 0) \overset{\sim}{\Rightarrow} (\ell_5, [y \mapsto 0 + 1], [], x > 0)}_{\textit{assign rule}}$$

$(\ell_5, [y \mapsto 1], [], x > 0)$

The subtree of s is safe if:

- s is not an error state
- If $s \overset{\sim}{\Rightarrow} s'$, one of these must hold:
  - s' is infeasible
  - There a child s'' of s such that:
    - *s' and s'' are equivalent*
    - The subtree of s'' is safe

```
Lemma L4: (safe t4).
Proof.
 apply Safe.
 - apply not_error_assign.
 - intros s' Hstep.
   left.
   exists t5.
   split.
   + apply in_eq.
   + split.
     {
       inversion Hstep.
       ...
     }
     { apply L5.}
Qed.
```

$$\underbrace{(\ell_4, [y \mapsto 0], [], x > 0) \overset{\sim}{\Rightarrow} (\ell_5, [y \mapsto 0 + 1], [], x > 0)}_{\textit{assign rule}}$$

$(\ell_5, [y \mapsto 1], [], x > 0)$

The subtree of s is safe if:

- s is not an error state
- If $s \overset{\sim}{\Rightarrow} s'$, one of these must hold:
  - s' is infeasible
  - There a child s'' of s such that:
    - *s' and s'' are equivalent*
    - The subtree of s'' is safe

```
Lemma L4: (safe t4).
Proof.
 apply Safe.
 - apply not_error_assign.
 - intros s' Hstep.
   left.
   exists t5.
   split.
   + apply in_eq.
   + split.
     {
       inversion Hstep.
       ...
     }
     { apply L5.}
Qed.
```

$$\underbrace{(\ell_4, [y \mapsto 0], [], x > 0) \overset{\sim}{\Rightarrow} (\ell_5, [y \mapsto 0 + 1], [], x > 0)}_{\textit{assign rule}}$$

$(\ell_5, [y \mapsto 1], [], x > 0)$

The subtree of s is safe if:

- s is not an error state
- If $s \overset{\sim}{\Rightarrow} s'$, one of these must hold:
  - s' is infeasible
  - There a child s'' of s such that:
    - *s' and s'' are equivalent*
    - The subtree of s'' is safe

```
Lemma L4: (safe t4).
Proof.
 apply Safe.
 - apply not_error_assign.
 - intros s' Hstep.
   left.
   exists t5.
   split.
   + apply in_eq.
   + split.
     {
       inversion Hstep.
       ...
     }
     { apply L5.}
Qed.
```

$$\underbrace{(\ell_4, [y \mapsto 0], [], x > 0) \overset{\sim}{\Rightarrow} (\ell_5, [y \mapsto 0 + 1], [], x > 0)}_{\textit{assign rule}}$$

$(\ell_5, [y \mapsto 1], [], x > 0)$

The subtree of s is safe if:

- s is not an error state
- If $s \overset{\sim}{\Rightarrow} s'$, one of these must hold:
  - s' is infeasible
  - There a child s'' of s such that:
    - *s' and s'' are equivalent*
    - The subtree of s'' is safe

```
Lemma L4: (safe t4).
Proof.
 apply Safe.
 - apply not_error_assign.
 - intros s' Hstep.
   left.
   exists t5.
   split.
   + apply in_eq.
   + split.
     {
       inversion Hstep.
       ...
     }
     { apply L5.}
Qed.
```

$$\underbrace{(\ell_4, [y \mapsto 0], [], x > 0) \overset{\sim}{\Rightarrow} (\ell_5, [y \mapsto 0 + 1], [], x > 0)}$$

assign rule

$(\ell_5, [y \mapsto 1], [], x > 0)$

The subtree of s is safe if:

- s is not an error state
- If $s \overset{\sim}{\Rightarrow} s'$, one of these must hold:
  - s' is infeasible
  - There a child s'' of s such that:
    - *s' and s'' are equivalent*
    - The subtree of s'' is safe

```
Lemma L4: (safe t4).
Proof.
 apply Safe.
 - apply not_error_assign.
 - intros s' Hstep.
   left.
   exists t5.
   split.
   + apply in_eq.
   + split.
     {
       inversion Hstep.
       ...
     }
     { apply L5.}
Qed.
```

long proof
expensive tactics

$$\underbrace{(\ell_4, [y \mapsto 0], [], x > 0) \overset{\sim}{\Rightarrow} (\ell_5, [y \mapsto 0 + 1], [], x > 0)}_{\textit{assign rule}}$$

$(\ell_5, [y \mapsto 1], [], x > 0)$

Lemma for *assign* $(x = e)$:
Let $s \overset{\text{def}}{=} (l, \tilde{\sigma}, \tilde{\kappa}, \varphi)$, if there exist t and $\tilde{\sigma}_{opt}$ such that:

- $\tilde{\sigma}\left[x \mapsto \widetilde{eval}(\tilde{\sigma}, e)\right] \equiv \tilde{\sigma}_{opt}$
- $t.state = (next(l), \tilde{\sigma}_{opt}, \tilde{\kappa}, \varphi)$
- t is safe

then $tree(s, [t])$ is safe.

```
Lemma L4: (safe t4).
Proof.
  apply (safe_assign ...).
  - apply (equiv_store ...).
  - reflexivity.
  - apply L5.
Qed.
```

smaller proof
simple tactics

```
Lemma safe_assign : …
Proof.
…
…
…
…
Qed.
```

expensive tactics
proved only once

Implementation: *KLEE-Rocq*

- Formalization in Rocq *(roughly 20K lines)*
  - Subset of LLVM with integers
  - LLVM semantics *(concrete and symbolic)*
  - Theory *(lemmas and theorems)*
- Proof generation on top of KLEE
  - Extended to generate a proof (.v file)

Evaluation

- Benchmarks
  - SVComp
  - WiSE (FSE'23)
  - Various numerical algorithms
- Metrics
  - Runtime overhead of proof generation
  - Proof compilation time
  - Proof size

	Base	Proof-Generation							#Paths	#Inst
	Analysis Time (seconds)	Analysis Time (seconds)	Proof Time (seconds)			Proof Size (KBs)				
			PG_{opt}	PG_{ltac}	PG_{none}	PG_{opt}	PG_{ltac}	PG_{none}		
svcomp_gcd1	0.1	0.1	1.4	1.9	4.5	419	476	1416	4	82
svcomp_gcd2	2.6	2.7	9.1	95.3	159.6	3242	27373	40743	42	1026
svcomp_gcd3	2.9	3.0	9.3	96.1	158.6	3269	27441	40897	45	1035
svcomp_sum	0.1	0.1	1.7	2.1	5.9	523	579	2329	10	158
svcomp_modulus	288.8	291.0	562.9	*OOM*	*OOM*	44517	-	-	398	15082
svcomp_num1	0.1	0.1	2.8	3.3	9.6	959	1064	3924	1	227
svcomp_num2	0.7	1.3	102.3	181.4	522.7	38983	42780	191914	256	11744
svcomp_bits	0.1	0.1	6.4	18.2	50.2	2358	7425	16529	1	650
svcomp_byte_add	0.4	1.1	142.2	263.8	482.6	70197	110753	192877	28	6239
WiSE_gcd	115.2	120.0	250.4	*OOM*	*OOM*	85845	-	-	1507	29228
WiSE_factorial	0.1	0.1	7.2	16.7	47.4	2967	3356	18903	13	1109
WiSE_sqrt	6.4	6.5	15.6	111.4	172.7	4656	10764	32428	91	1735
jenkins	30.9	31.1	1.0	1.4	2.9	231	340	702	2	36
murmur	35.4	35.3	1.0	1.3	2.7	237	294	631	2	36
fnv1a	2037.2	2049.3	0.9	0.9	1.5	171	175	327	2	18
is_prime	1.2	1.3	16.0	38.1	128.1	7075	8308	44624	30	2553
reverse	0.1	0.1	1.9	48.0	58.6	315	15274	15973	1	59

Summary & Future Work

- Generating safety proofs using symbolic execution
  - Formal framework in Rocq
  - Runtime proof generation
- Applied for LLVM/KLEE
- Support advanced LLVM features
- Can be potentially applied to:
  - Other symbolic execution tools
  - Other program analysis techniques

https://github.com/davidtr1037/klee-rocq