How to Win SV-COMP with Symbolic Execution

Jan Strejček
Masaryk University
Brno, Czech Republic

KLEE Workshop 2022

SV-COMP

SV-COMP = Competition on Software Verification

- organized by Dirk Beyer since 2012
- task = to decide whether a given C (or Java) program satisfies a given property (and produce a witness)
- considered properties
  - reachability safety
  - memory safety
  - no overflows
  - termination
- resources: 8 cores, 900 s of CPU time, 15 GB of memory

SV-COMP

SV-COMP = Competition on Software Verification

- organized by Dirk Beyer since 2012
- task = to decide whether a given C (or Java) program satisfies a given property (and produce a witness)
- considered properties
  - reachability safety
  - memory safety
  - no overflows
  - termination
- resources: 8 cores, 900 s of CPU time, 15 GB of memory

SV-COMP 2022

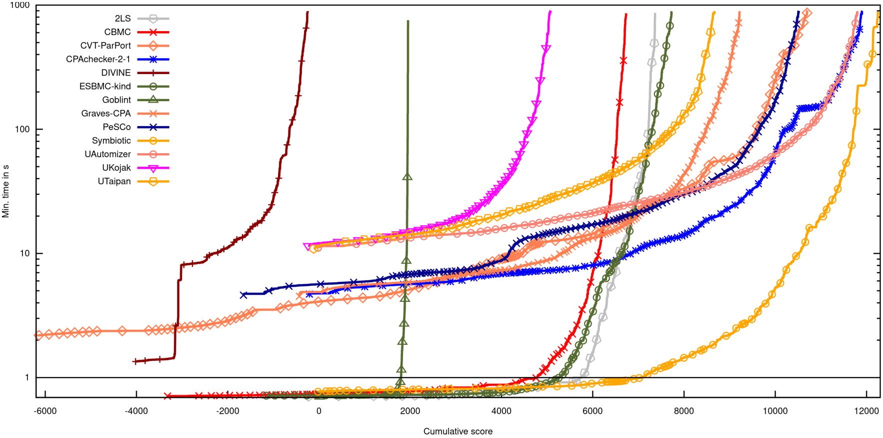
- 15 648 verification tasks
- 40 verification tools (including 12 hours concours)
- 12 of them use symbolic execution

Symbiotic at SV-COMP

- participating since 2013 (every year except 2015)
- 4 gold medals in MemSafety (2018, 2019, 2021, 2022)
- 3 gold medals in SoftwareSystems (2020, 2021, 2022)
- overall winner of SV-COMP 2022

Symbiotic at SV-COMP

- participating since 2013 (every year except 2015)
- 4 gold medals in MemSafety (2018, 2019, 2021, 2022)
- 3 gold medals in SoftwareSystems (2020, 2021, 2022)
- overall winner of SV-COMP 2022



source: `https://sv-comp.sosy-lab.org/2022/results/results-verified/`

why Symbiotic uses symbolic execution

Jiří Slabý

Marek Trtík

why SYMBIOTIC uses symbolic execution

Jiří Slabý

Marek Trtík

pros

+ no false alarms
+ KLEE is available
+ KLEE easily finds bugs

why SYMBIOTIC uses symbolic execution

Jiří Slabý

Marek Trtík

pros

+ no false alarms
+ KLEE is available
+ KLEE easily finds bugs

cons

– path explosion problem
– struggles with program loops
– rarely finishes on real programs
– KLEE skips some runs

why SYMBIOTIC uses symbolic execution

Jiří Slabý

Marek Trtík

Marek Chalupa

pros

+ no false alarms
+ KLEE is available
+ KLEE easily finds bugs

cons

– path explosion problem
– struggles with program loops
– rarely finishes on real programs
– KLEE skips some runs

outline

1. how Symbiotic works
   - Chalupa and Strejček: *Symbiotic: Slice and Verify*. Under review.
   - Chalupa, Mihalkovič, Řechtáčková, Zaoral, and Strejček: *Symbiotic 9: String Analysis and Backward Symbolic Execution with Loop Folding*. TACAS 2022.

outline

1. how SYMBIOTIC works
   - Chalupa and Strejček: *Symbiotic: Slice and Verify*. Under review.
   - Chalupa, Mihalkovič, Řechtáčková, Zaoral, and Strejček: *Symbiotic 9: String Analysis and Backward Symbolic Execution with Loop Folding*. TACAS 2022.
2. what SLOWBEAST does
   - backward symbolic execution (BSE) $=$ k-induction
   - BSE $+$ loop folding (BSELF)
   - Chalupa and Strejček: *Backward Symbolic Execution with Loop Folding*. SAS 2021.

outline

1. how Symbiotic works
   - Chalupa and Strejček: *Symbiotic: Slice and Verify*. Under review.
   - Chalupa, Mihalkovič, Řechtáčková, Zaoral, and Strejček: *Symbiotic 9: String Analysis and Backward Symbolic Execution with Loop Folding*. TACAS 2022.
2. what Slowbeast does
   - backward symbolic execution (BSE) = k-induction
   - BSE + loop folding (BSELF)
   - Chalupa and Strejček: *Backward Symbolic Execution with Loop Folding*. SAS 2021.

JetKlee

- our fork of Klee optimized for verification
- analysis of all possible runs is more important than speed
- `https://github.com/staticafi/JetKlee`

	Klee	JetKlee
symbolic pointers	✓	✓
symbolic-sized allocations	✗	✓
symbolic addresses	✗	✓

JetKlee and Slowbeast

JetKlee

- our fork of Klee optimized for verification
- analysis of all possible runs is more important than speed
- https://github.com/staticafi/JetKlee

Slowbeast

- symbolic executor implemented by Marek Chalupa in Python
- https://gitlab.fi.muni.cz/xchalup4/slowbeast

	Klee	JetKlee	Slowbeast
symbolic pointers	✓	✓	✓
symbolic-sized allocations	✗	✓	✓
symbolic addresses	✗	✓	✓
symbolic floats	✗	✗	✓
parallel programs	✗	✗	✓
backward symbolic exec. (BSE)	✗	✗	✓
BSE + loop folding (BSELF)	✗	✗	✓
invariant generation	✗	✗	✓

program slicing

```
n = input();
i = 0;
while (i != n) {
  c = input();
  if (i == 0) {
    min = c;
    max = c;
  }
  if (c < min)
    min = c;
  if (c > max)
    max = c;
  i = i + 2;
}
assert(min <= c);
assert(even(n));
```

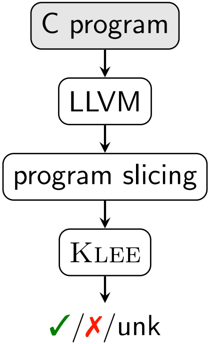
program slicing

```
n = input();
i = 0;
while (i != n) {
  c = input();
  if (i == 0) {
    min = c;
    max = c;
  }
  if (c < min)
    min = c;
  if (c > max)
    max = c;
  i = i + 2;
}
assert(min <= c);
assert(even(n));
```

```
n = input();
i = 0;
while (i != n) {
  c = input();
  if (i == 0) {
    min = c;
    max = c;
  }
  if (c < min)
    min = c;
  if (c > max)
    max = c;
  i = i + 2;
}
assert(min <= c);
assert(even(n));
```

```
n = input();
i = 0;
while (i != n) {
  c = input();
  if (i == 0) {
    min = c;
    max = c;
  }
  if (c < min)
    min = c;
  if (c > max)
    max = c;
  i = i + 2;
}
assert(min <= c);
assert(even(n));
```

first workflow of SYMBIOTIC for reachability safety



program slicing

```
n = input();
i = 0;
while (i != n) {
  c = input();
  if (i == 0) {
    min = c;
    max = c;
  }
  if (c < min)
    min = c;
  if (c > max)
    max = c;
  i = i + 2;
}
assert(min <= c);
assert(even(n));
```

program slicing

```
n = input();
i = 0;
while (i != n) {
  c = input();
  if (i == 0) {
    min = c;
    max = c;
  }
  if (c < min)
    min = c;
  if (c > max)
    max = c;
  i = i + 2;
}
assert(min <= c);
assert(even(n));
```

program slicing

```
n = input();
i = 0;
while (i != n) {
  c = input();
  if (i == 0) {
    min = c;
    max = c;
  }
  if (c < min)
    min = c;
  if (c > max)
    max = c;
  i = i + 2;
}
assert(min <= c);
assert(even(n));
```

```
n = input();
i = 0;
while (i != n) {
  c = input();
  if (i == 0) {
    min = c;
    max = c;
  }
  if (c < min)
    min = c;
  if (c > max)
    max = c;
  i = i + 2;
}
assert(min <= c);
assert(even(n));
```

program slicing

```
n = input();
i = 0;
while (i != n) {
  c = input();
  if (i == 0) {
    min = c;
    max = c;
  }
  if (c < min)
    min = c;
  if (c > max)
    max = c;
  i = i + 2;
}
assert(min <= c);
assert(even(n));
```

```
n = input();
i = 0;
while (i != n) {
  c = input();
  if (i == 0) {
    min = c;
    max = c;
  }
  if (c < min)
    min = c;
  if (c > max)
    max = c;
  i = i + 2;
}
assert(min <= c);
assert(even(n));
```

```
n = input();
i = 0;
while (i != n) {
  c = input();
  if (i == 0) {
    min = c;
    max = c;
  }
  if (c < min)
    min = c;
  if (c > max)
    max = c;
  i = i + 2;
}
assert(min <= c);
assert(even(n));
```

program slicing

```
n = input();
i = 0;
while (i != n) {
  c = input();
  if (i == 0) {
    min = c;
    max = c;
  }
  if (c < min)
    min = c;
  if (c > max)
    max = c;
  i = i + 2;
}
assert(min <= c);
assert(even(n));
```

```
n = input();
i = 0;
while (i != n) {
  c = input();
  if (i == 0) {
    min = c;
    max = c;
  }
  if (c < min)
    min = c;
  if (c > max)
    max = c;
  i = i + 2;
}
assert(min <= c);
assert(even(n));
```

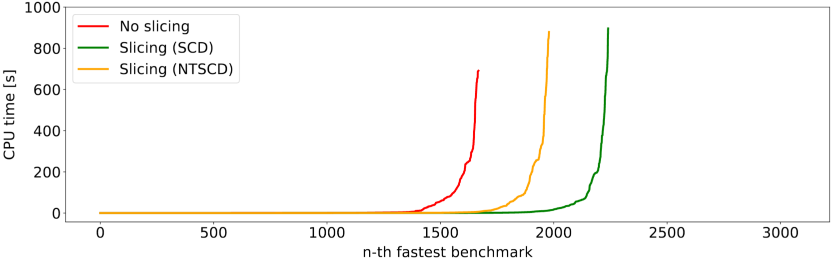
standard control dependence (SCD)

```
n = input();
i = 0;
while (i != n) {
  c = input();
  if (i == 0) {
    min = c;
    max = c;
  }
  if (c < min)
    min = c;
  if (c > max)
    max = c;
  i = i + 2;
}
assert(min <= c);
assert(even(n));
```

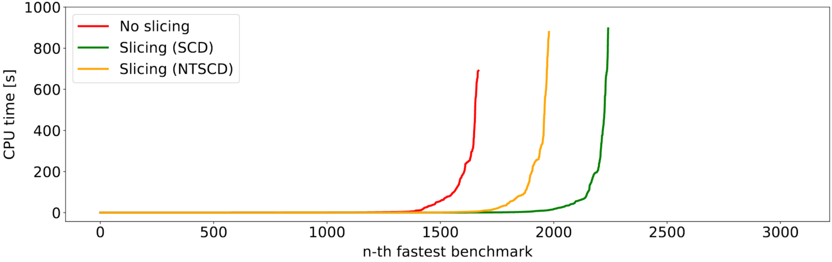
non-termination sensitive control dependence (NTSCD)

influence of slicing on performance of KLEE

correct verification results produced by KLEE with slicing on reachability safety tasks of SV-COMP 2019

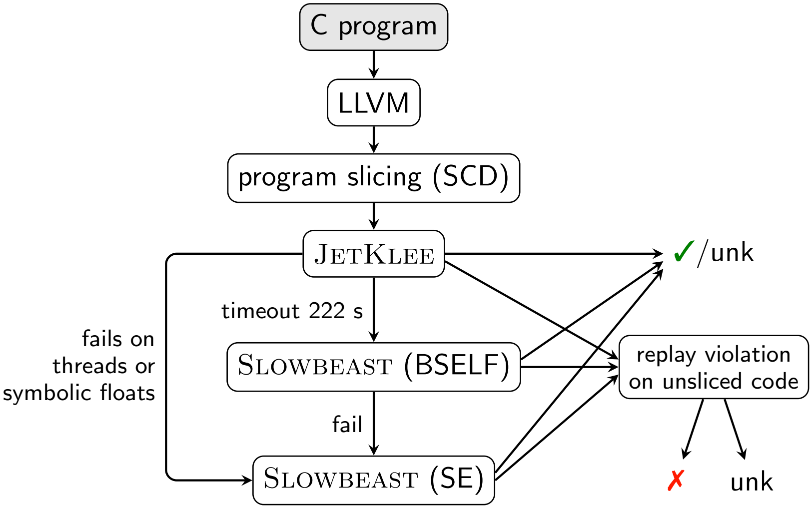


correct verification results produced by KLEE with slicing on reachability safety tasks of SV-COMP 2019



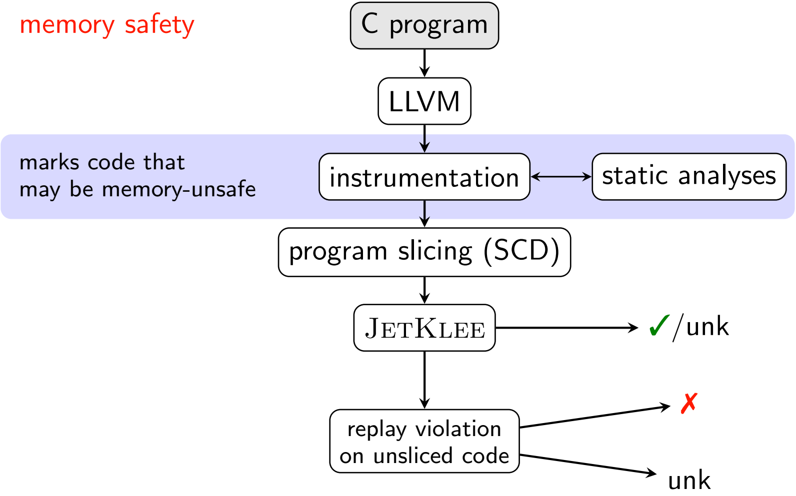
- slicing (SCD) also brought 43 incorrect verification results ✗
- Chalupa and Strejček: *Evaluation of Program Slicing in Software Verfication*. iFM 2019.

current workflow of Symbiotic for reachability safety

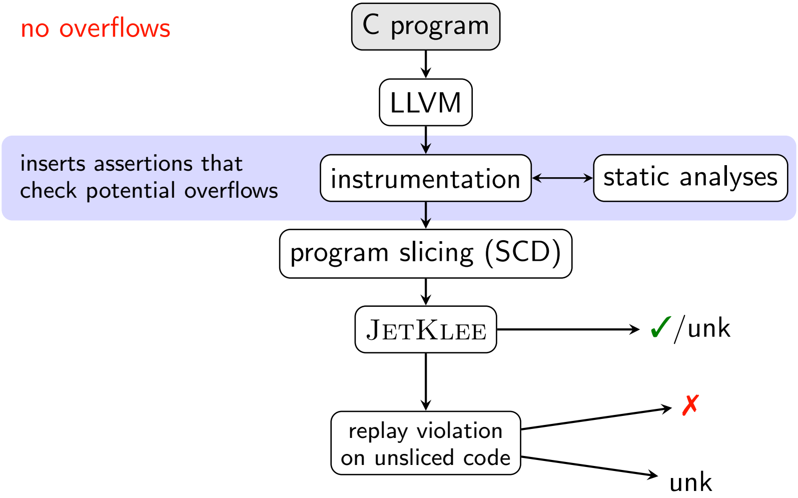


current workflow of Symbiotic for other properties

memory safety

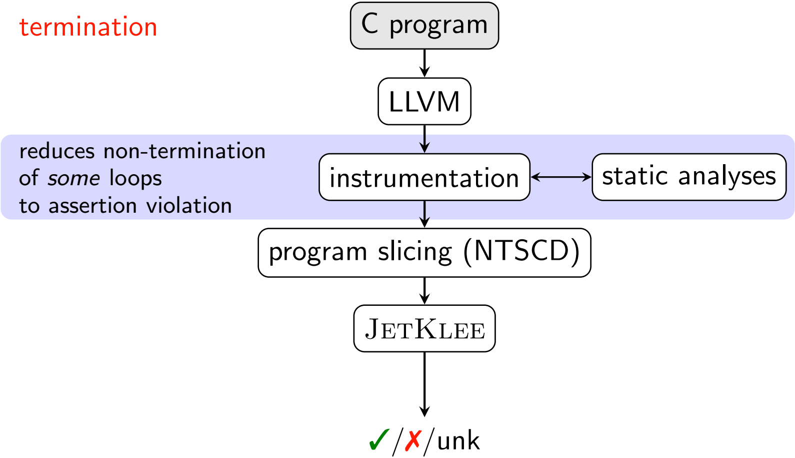


current workflow of SYMBIOTIC for other properties

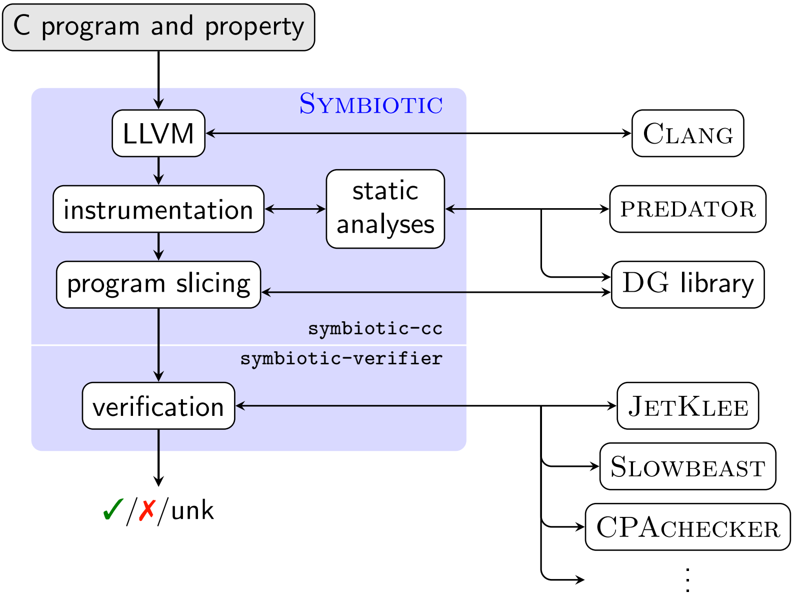


current workflow of Symbiotic for other properties

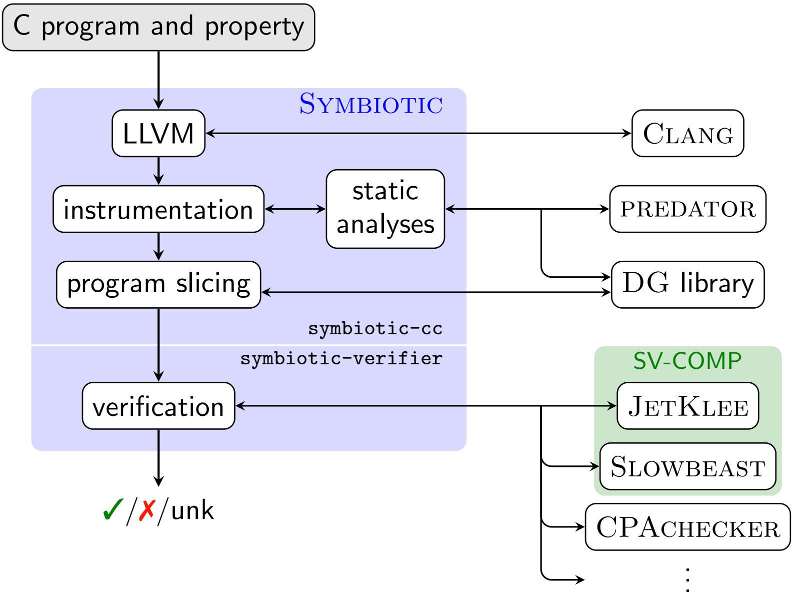
termination



general structure of Symbiotic



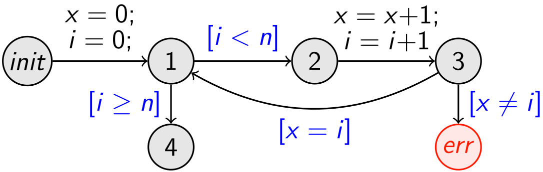
general structure of SYMBIOTIC



outline

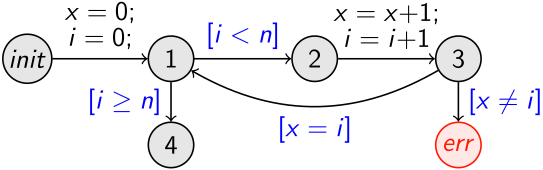
1. how SYMBIOTIC works
   - Chalupa and Strejček: *Symbiotic: Slice and Verify*. Under review.
   - Chalupa, Mihalkovič, Řechtáčková, Zaoral, and Strejček: *Symbiotic 9: String Analysis and Backward Symbolic Execution with Loop Folding*. TACAS 2022.
2. what SLOWBEAST does
   - backward symbolic execution (BSE) = k-induction
   - BSE + loop folding (BSELF)
   - Chalupa and Strejček: *Backward Symbolic Execution with Loop Folding*. SAS 2021.

```
int n; // input
int x = 0;
int i = 0;
while (i < n) {
  ++x;
  ++i;
  assert(x == i);
}
```



control flow automata (CFA)

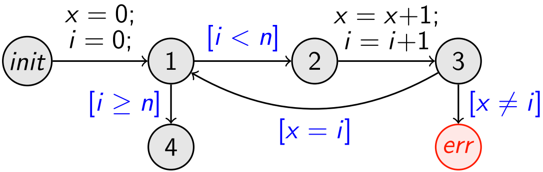
```
int n; // input
int x = 0;
int i = 0;
while (i < n) {
  ++x;
  ++i;
  assert(x == i);
}
```



- *err* has no successors
- a path is feasible if it can be entirely executed

control flow automata (CFA)

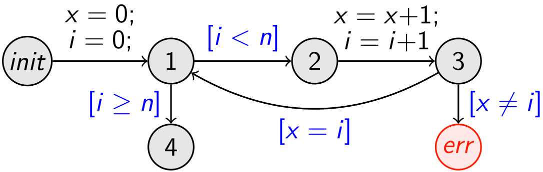
```
int n; // input
int x = 0;
int i = 0;
while (i < n) {
  ++x;
  ++i;
  assert(x == i);
}
```



- *err* has no successors
- a path is feasible if it can be entirely executed
- a path is unsafe if it is feasible and ends in *err*, it is safe otherwise

control flow automata (CFA)

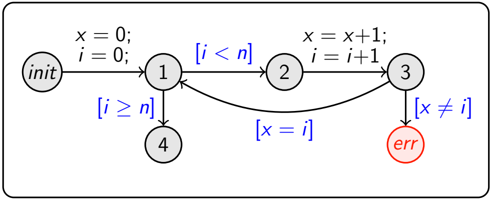
```
int n; // input
int x = 0;
int i = 0;
while (i < n) {
  ++x;
  ++i;
  assert(x == i);
}
```



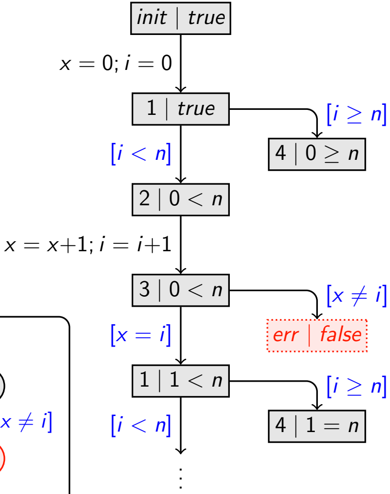
- *err* has no successors
- a path is feasible if it can be entirely executed
- a path is unsafe if it is feasible and ends in *err*, it is safe otherwise
- a CFA is correct if all paths starting in *init* are safe, it is incorrect otherwise

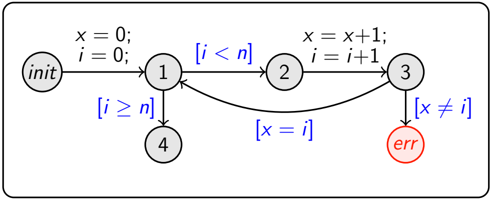
symbolic execution (SE)

```
int n; // input
int x = 0;
int i = 0;
while (i < n) {
  ++x;
  ++i;
  assert(x == i);
}
```



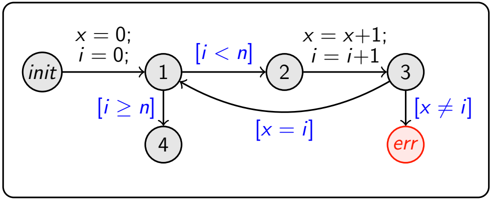
```
int n; // input
int x = 0;
int i = 0;
while (i < n) {
  ++x;
  ++i;
  assert(x == i);
}
```





backward symbolic execution (BSE)

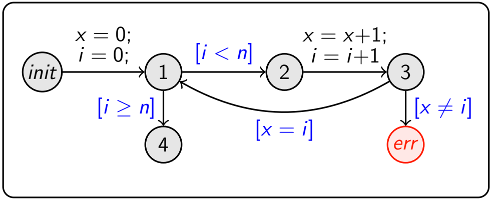
```
int n; // input
int x = 0;
int i = 0;
while (i < n) {
  ++x;
  ++i;
  assert(x == i);
}
```



backward symbolic execution (BSE)

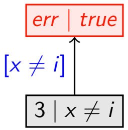
err | *true*

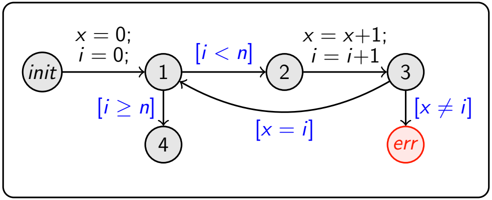
```
int n; // input
int x = 0;
int i = 0;
while (i < n) {
  ++x;
  ++i;
  assert(x == i);
}
```



backward symbolic execution (BSE)

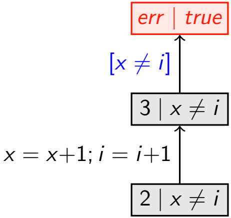
```
int n; // input
int x = 0;
int i = 0;
while (i < n) {
  ++x;
  ++i;
  assert(x == i);
}
```

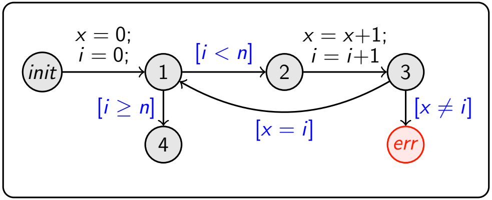




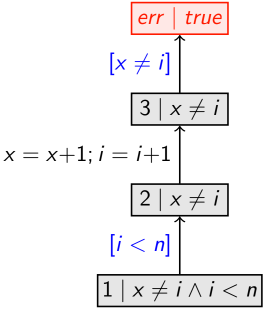
backward symbolic execution (BSE)

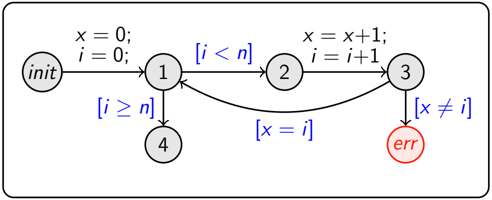
```
int n; // input
int x = 0;
int i = 0;
while (i < n) {
  ++x;
  ++i;
  assert(x == i);
}
```





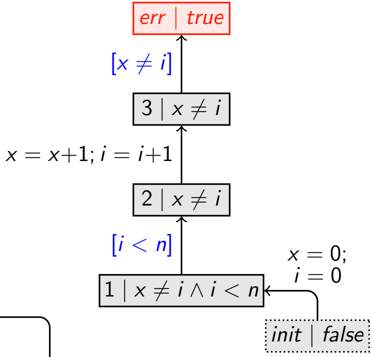
```
int n; // input
int x = 0;
int i = 0;
while (i < n) {
  ++x;
  ++i;
  assert(x == i);
}
```

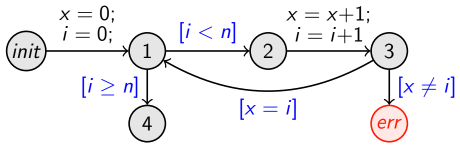




backward symbolic execution (BSE)

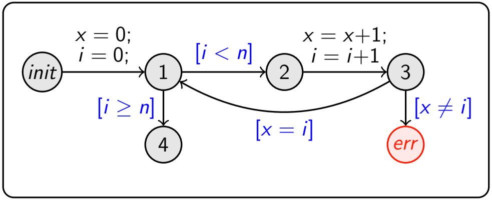
```
int n; // input
int x = 0;
int i = 0;
while (i < n) {
  ++x;
  ++i;
  assert(x == i);
}
```

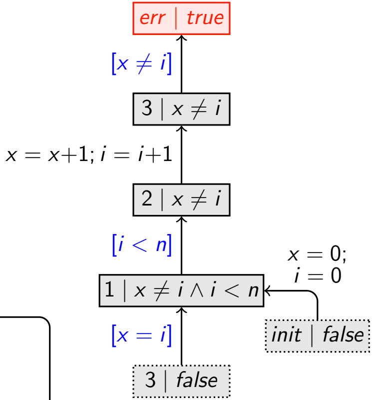




backward symbolic execution (BSE)

```
int n; // input
int x = 0;
int i = 0;
while (i < n) {
  ++x;
  ++i;
  assert(x == i);
}
```





k-induction for CFA

k-induction for CFA

A CFA is correct if the following holds for some $k > 0$.

1. base case
   All paths of length at most k starting in *init* are safe.
2. induction step
   Each path of length $k + 1$ that has a safe prefix of length k is also safe.

k-induction for CFA

k-induction for CFA

A CFA is correct if the following holds for some $k > 0$.

1. base case
   All paths of length at most k starting in *init* are safe.
2. induction step
   Each path of length $k+1$ that has a safe prefix of length k is also safe.

verification algorithm

1. $k \leftarrow 1$
2. <u>if</u> base case does not hold <u>then</u> <u>return</u> incorrect
3. <u>if</u> induction step holds <u>then</u> <u>return</u> correct
4. $k \leftarrow k + 1$
5. <u>goto</u> 2

relating *k*-induction and BSE

base case

All paths of length at most *k* starting in *init* are safe.

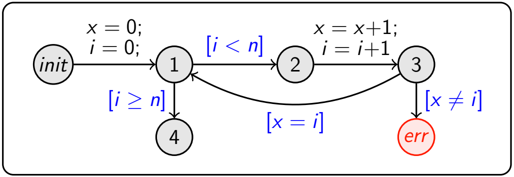
- i.e. there is no feasible path from *init* to *err* of length at most *k*

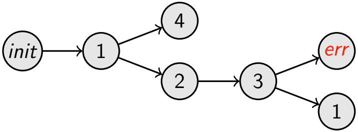
relating k-induction and BSE

base case

All paths of length at most k starting in *init* are safe.

- i.e. there is no feasible path from *init* to *err* of length at most k
- we can either search all relevant paths starting in *init*



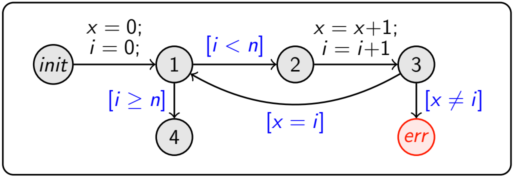


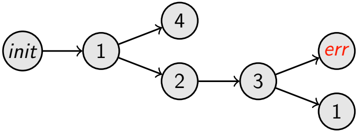
relating k-induction and BSE

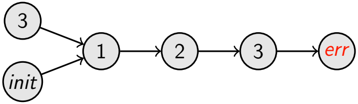
base case

All paths of length at most k starting in *init* are safe.

- i.e. there is no feasible path from *init* to *err* of length at most k
- we can either search all relevant paths starting in *init*
- or search all relevant paths leading to *err*







relating k-induction and BSE

induction step

Each path of length $k + 1$ that has a safe prefix of length k is also safe.

- i.e. there is no unsafe path of length $k + 1$ with a safe prefix of length k
- but a proper prefix of each unsafe path is safe
- i.e. there is no feasible path to *err* of length $k + 1$
- i.e. the BSE tree is finite

relating k-induction and BSE

Theorem (BSE = k-induction)

If a CFA is incorrect, then the k-induction algorithm detects it and BSE tree will contain an unsafe path from init.
If a CFA is correct, then k-induction algorithm detects it if and only if the BFS tree is finite and contains no init node.

Both approaches fail to detect correctness of a CFA that contains an unsafe path of length k for each $k > 0$ (i.e. BSE tree is infinite).

BSE with Loop Folding (BSELF)

- BSE (and k-induction) is incomplete

```
int n; // input
int x = 0;
int i = 0;
while (i < n) {
  ++x;
  ++i;
  assert(x == i);
}
```

BSE with Loop Folding (BSELF)

- BSE (and k-induction) is incomplete

```
int n; // input
int x = 0;
int i = 0;
while (i < n) {
  ++x;
  ++i;

}
assert(x == i);
```

BSE with Loop Folding (BSELF)

- BSE (and k-induction) is incomplete
- invariants in loops can help

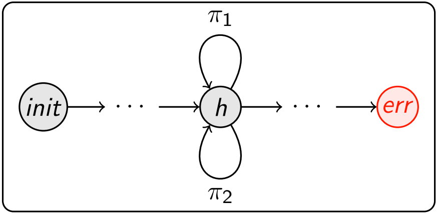
```
int n; // input
int x = 0;
int i = 0;
while (i < n) {
  ++x;
  ++i;

}
assert(x == i);
```

BSE with Loop Folding (BSELF)

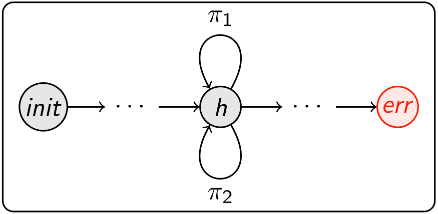
- BSE (and k-induction) is incomplete
- invariants in loops can help
- loop folding computes loop invariants from BSE states

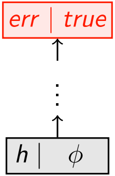
```
int n; // input
int x = 0;
int i = 0;
while (i < n) {
  ++x;
  ++i;

}
assert(x == i);
```

BSE with Loop Folding (BSELF)

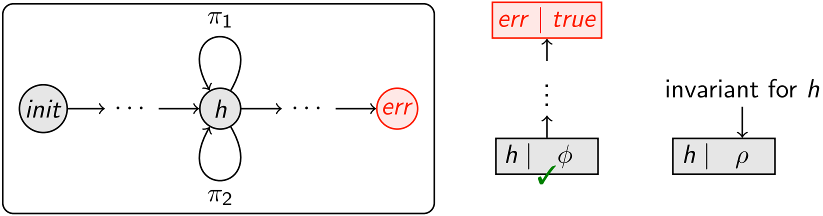


BSE with Loop Folding (BSELF)



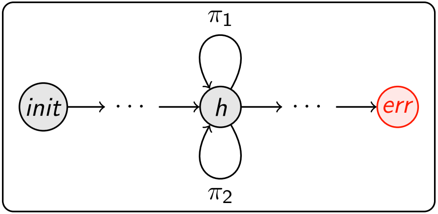


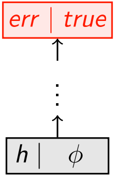
BSE with Loop Folding (BSELF)



- when BSE reaches a node $h \mid \phi$ where h is a loop header, we try to find an invariant ρ for h satisfying $\rho \implies \neg\phi$
- if we succeed, we can drop this path

BSE with Loop Folding (BSELF)

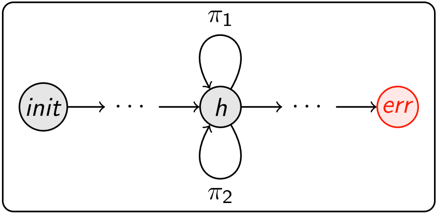


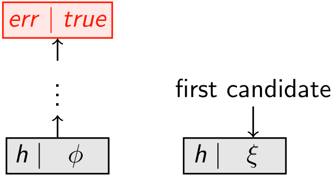


- we gradually create invarant candidates
- each candidate ξ satisfies $\xi \implies \neg\phi$ and is inductive, i.e.

  if $h \mid \xi$ $\longrightarrow \cdots \longrightarrow$ $h \mid \xi'$ then $\xi' \implies \xi$

BSE with Loop Folding (BSELF)



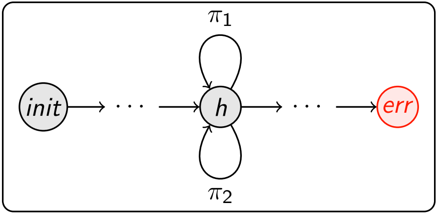


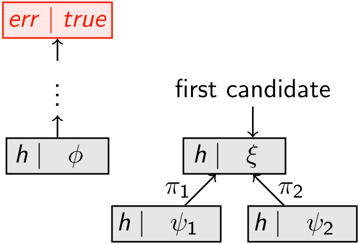
- we gradually create invarant candidates
- each candidate ξ satisfies $\xi \implies \neg\phi$ and is inductive, i.e.

  if [h | ξ] $\longrightarrow \cdots \longrightarrow$ [h | ξ'] then $\xi' \implies \xi$

1. find first invariant candidate ξ such that location h cannot be reached again from [h | ξ]

BSE with Loop Folding (BSELF)



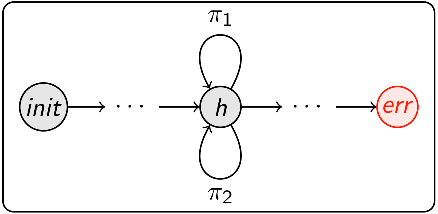


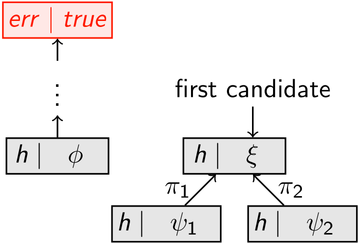
- we gradually create invarant candidates
- each candidate ξ satisfies $\xi \implies \neg\phi$ and is inductive, i.e.

  if $[h \mid \xi] \rightarrow \cdots \rightarrow [h \mid \xi']$ then $\xi' \implies \xi$

1. find first invariant candidate ξ such that location h cannot be reached again from $[h \mid \xi]$
2. if ξ is not an invariant, then compute ψ_1, ψ_2

BSE with Loop Folding (BSELF)





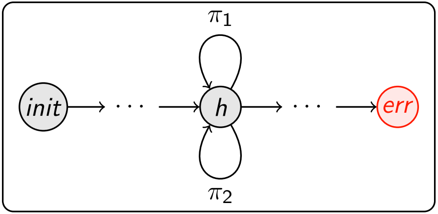
- we gradually create invarant candidates
- each candidate ξ satisfies $\xi \implies \neg\phi$ and is inductive, i.e.

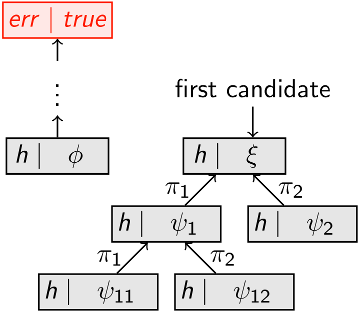
  if $\boxed{h \mid \xi} \longrightarrow \cdots \longrightarrow \boxed{h \mid \xi'}$ then $\xi' \implies \xi$

1. find first invariant candidate ξ such that location h cannot be reached again from $\boxed{h \mid \xi}$
2. if ξ is not an invariant, then compute ψ_1, ψ_2
3. if $\psi_i \implies \neg\phi$, then $\psi_i \vee \xi$ is also a candidate

   ⋮

BSE with Loop Folding (BSELF)



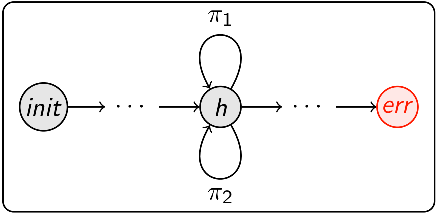


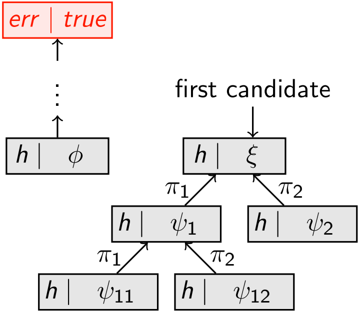
⋮

- candidates $\psi_{11} \vee \psi_1 \vee \xi$ and $\psi_{12} \vee \psi_1 \vee \xi$

⋮

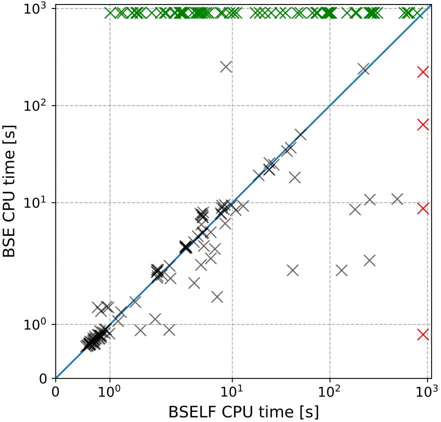
BSE with Loop Folding (BSELF)





- we also apply overapproximation to candidates
- searching for an invariant is restricted to not get stuck
- if invariant is not found, we continue with BSE
- but candidates are saved and used for the construction of the first candidate when we enter *h* next time

BSE vs. BSELF on reachability safety tasks
from the Loops subcategory of SV-COMP 2021
(only benchmarks solved by BSE or BSELF)



conclusion

to win SV-COMP with symbolic execution

- first use static analyses and slicing to reduce the program
- tune symbolic executor to handle various code features precisely
- combine SE with BSE and potentialy other techniques
- fix all bugs

conclusion

to win SV-COMP with symbolic execution

- first use static analyses and slicing to reduce the program
- tune symbolic executor to handle various code features precisely
- combine SE with BSE and potentialy other techniques
- fix all bugs

Thank you.