

ESBMC v6.0

Mikhail R. Gadelha
Felipe R. Monteiro
Lucas C. Cordeiro
Denis A. Nicole

25th Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems

8th Intl. Competition on Software Verification

ESBMC v6.0

Verifying C Programs Using k -Induction and Invariant Inference
(Competition Contribution)

Mikhail R. Gadelha, **Felipe R. Monteiro**, Lucas C. Cordeiro, and Denis A. Nicole



UNIVERSITY OF
Southampton



MANCHESTER
1824

ESBMC

Gadelha et al., ASE'18

- SMT-based bounded model checker of single- and multi-threaded C/C++ programs
 - turned 10 years old in 2018
- Combines BMC, k -induction and abstract interpretation:
 - path towards correctness proof
 - bug hunting
- Exploits SMT solvers and their background theories
 - optimized encodings for pointers, bit operations, unions, arithmetic over- and underflow, and floating-points

ESBMC

Gadelha et al., ASE'18

- SMT-based bounded model checker of single- and multi-threaded C/C++ programs

arithmetic under- and overflow

pointer safety

array bounds

division by zero

memory leaks

atomicity and order violations

deadlock

data race

user-specified assertions

built-in properties

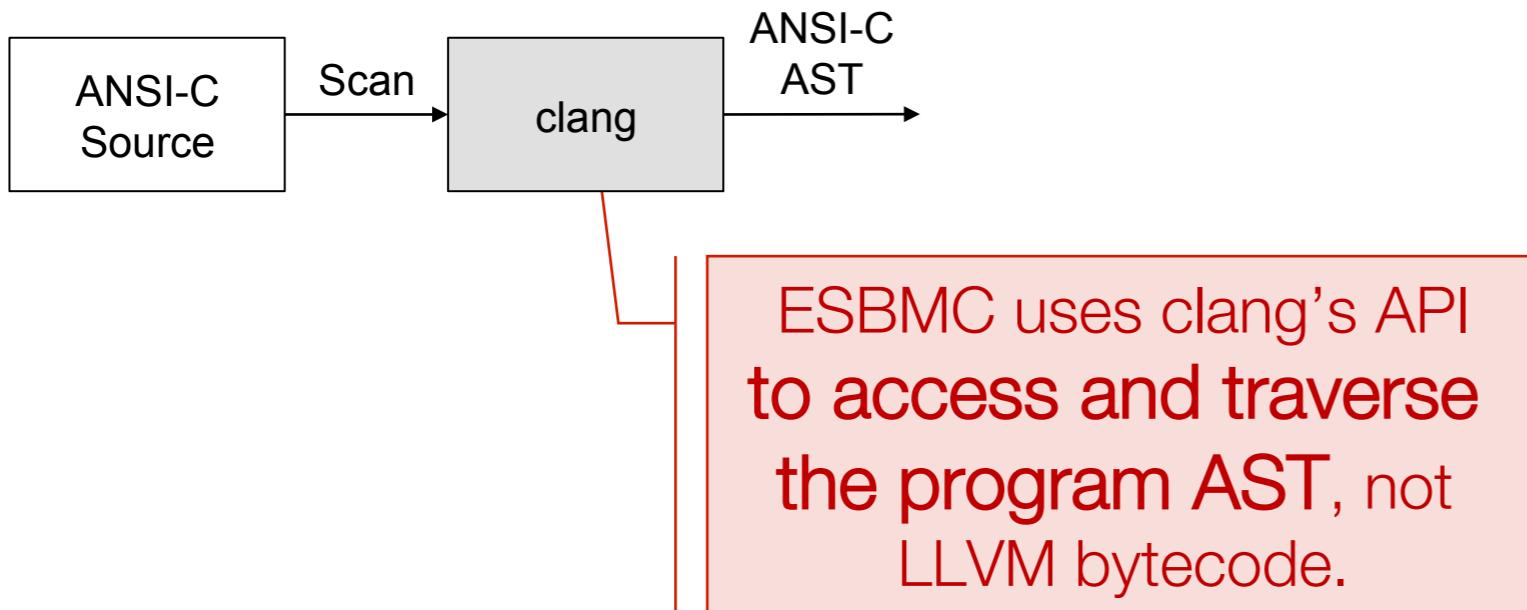
ESBMC Architecture

- ESBMC uses a k -induction proof rule to verify and falsify properties over C programs

ANSI-C
Source

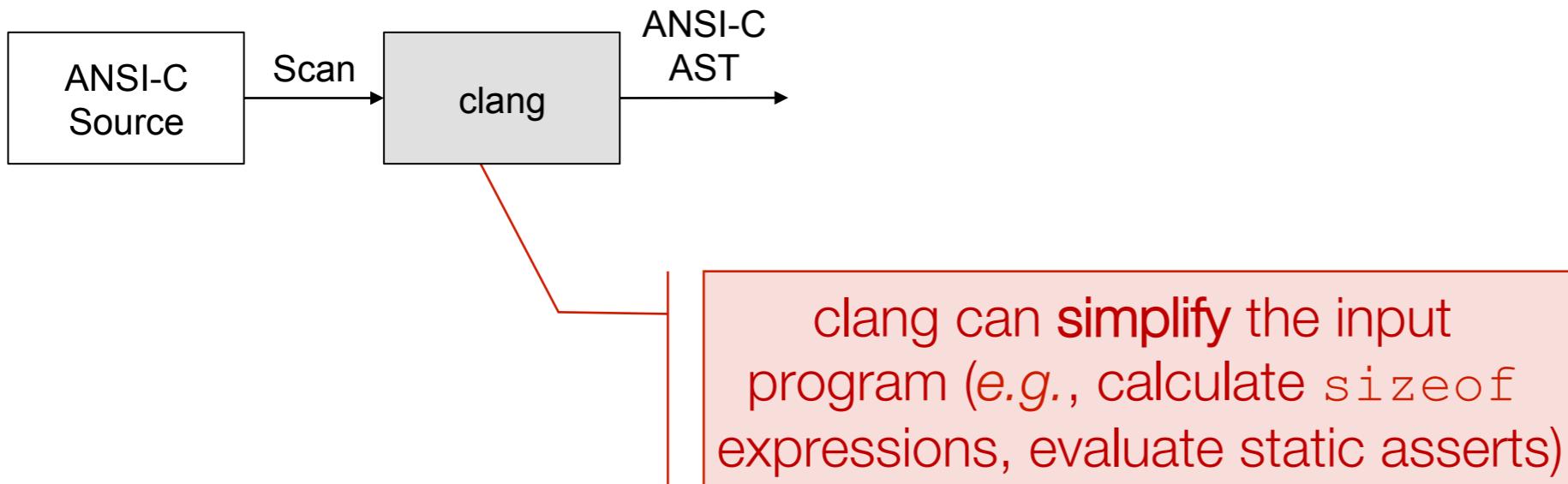
ESBMC Architecture

- ESBMC uses a k -induction proof rule to verify and falsify properties over C programs



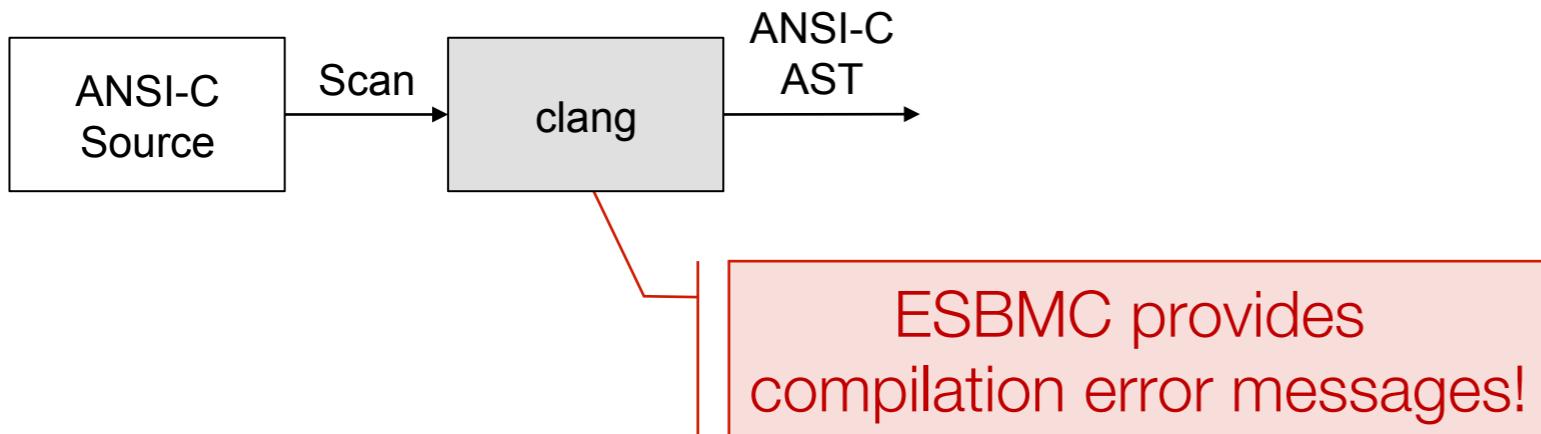
ESBMC Architecture

- ESBMC uses a k -induction proof rule to verify and falsify properties over C programs



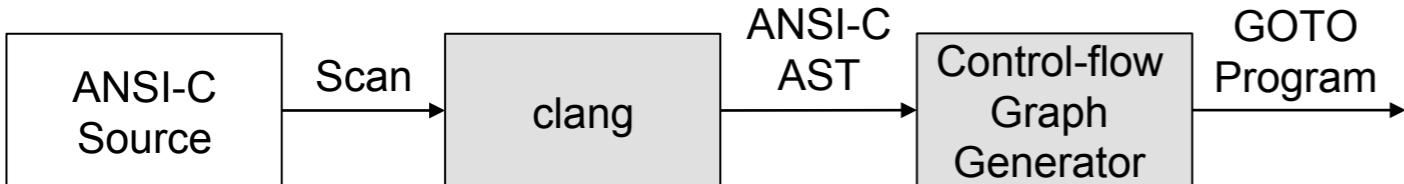
ESBMC Architecture

- ESBMC uses a k -induction proof rule to verify and falsify properties over C programs



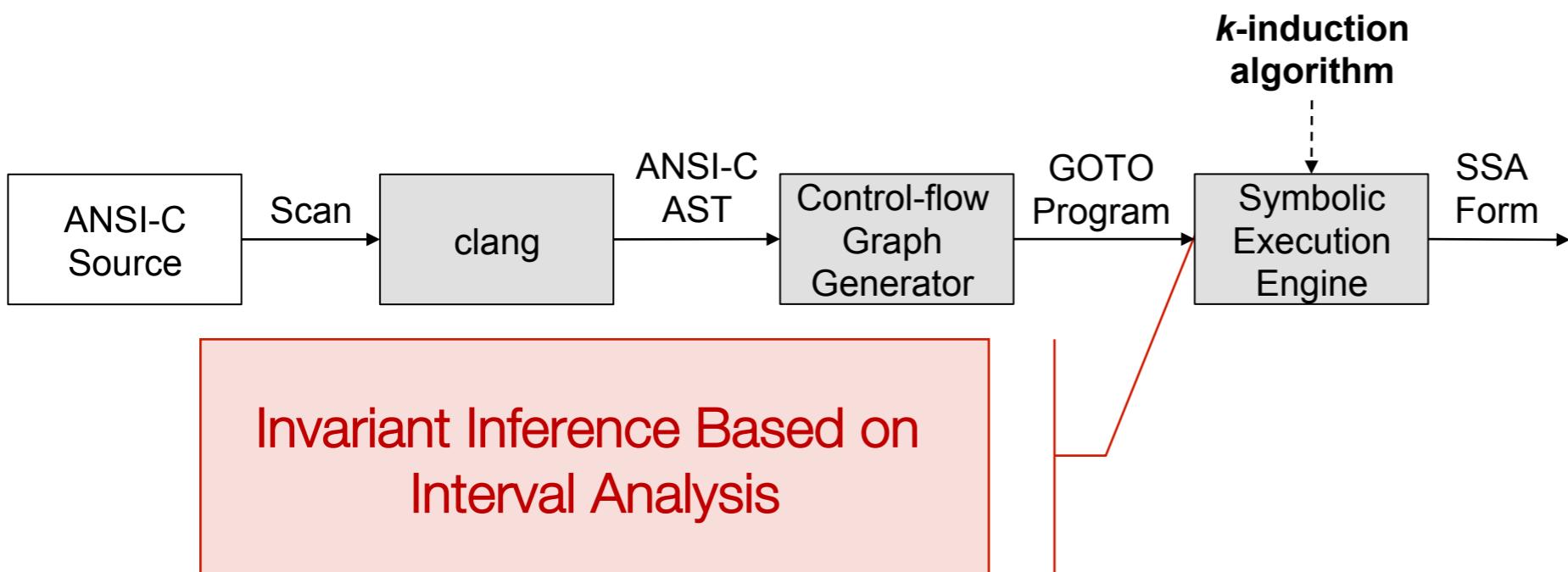
ESBMC Architecture

- The CFG generator takes the program AST and transforms it into an equivalent GOTO program
 - only of assignments, conditional and unconditional branches, assumes, and assertions.



ESBMC Architecture

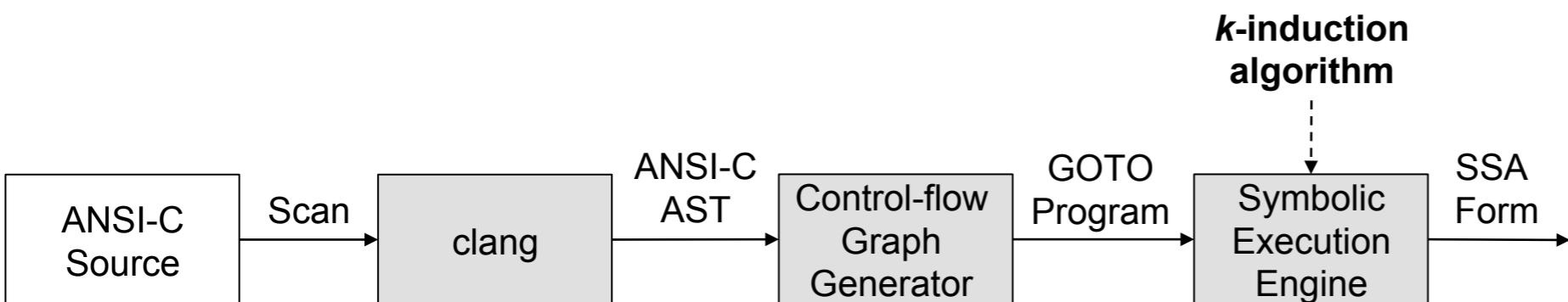
- ESBMC perform a static analysis prior to loop unwinding and (over-)estimate the range that a variable can assume
 - “rectangular” invariant generation based on interval analysis (e.g., $a \leq x \leq b$)



- Abstract-interpretation component from CPROVER
- Only for **integer** variables

ESBMC Architecture

- ESBMC perform a static analysis prior to loop unwinding and (over-)estimate the range that a variable can assume
 - “rectangular” invariant generation based on interval analysis (e.g., $a \leq x \leq b$)



International Journal on Software Tools for Technology Transfer
February 2017, Volume 19, Issue 1, pp 97–114 | Cite as

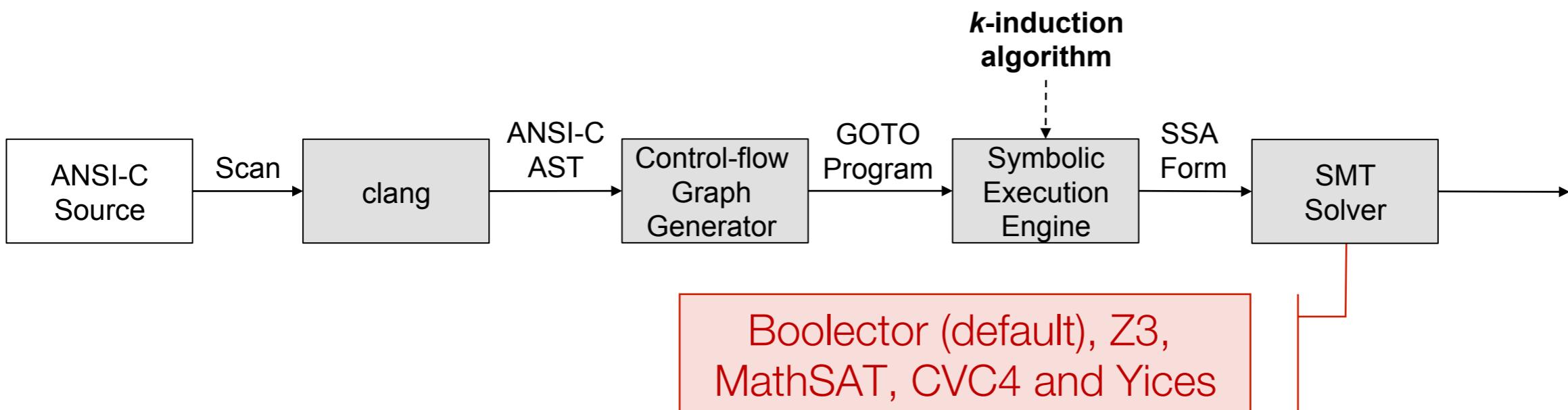
Handling loops in bounded model checking of C programs via *k*-induction

Authors
Mikhail Y. R. Gadelha, Hussama I. Ismail, Lucas C. Cordeiro

This block contains a screenshot of a journal article from the International Journal on Software Tools for Technology Transfer (STTT). The article is titled 'Handling loops in bounded model checking of C programs via *k*-induction'. It is published in February 2017, Volume 19, Issue 1, pages 97–114. The authors listed are Mikhail Y. R. Gadelha, Hussama I. Ismail, and Lucas C. Cordeiro. The screenshot also shows the Springer logo and the journal's title.

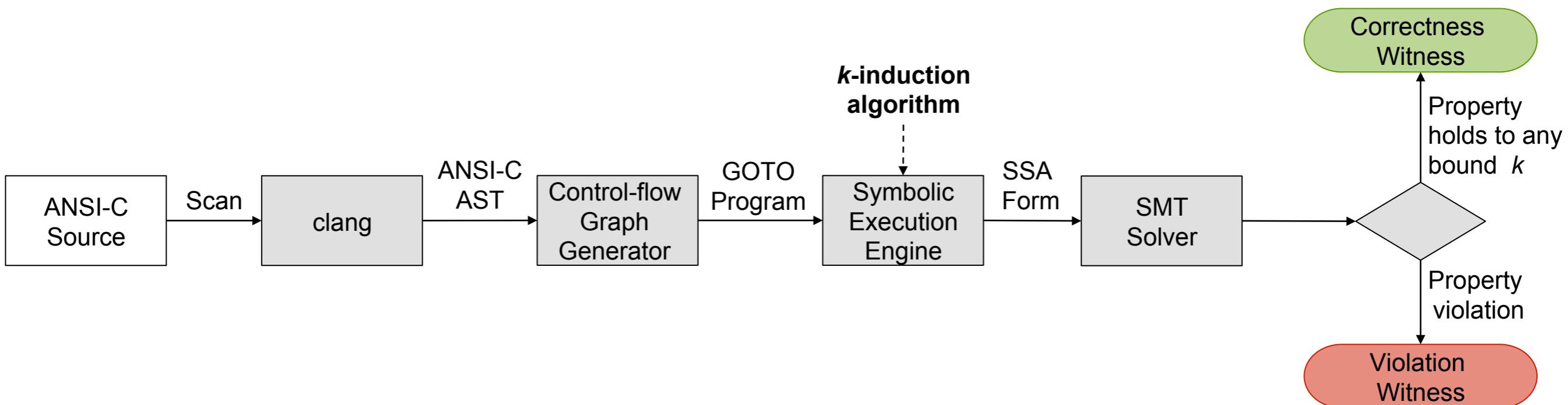
ESBMC Architecture

- The back-end is highly configurable and allows the encoding of quantifier-free formulas
bitvectors, arrays, tuple, fixed-point and floating-point arithmetic (all solvers), and linear integer and real arithmetic (all solvers but Boolector).



ESBMC Architecture

- The back-end is highly configurable and allows the encoding of quantifier-free formulas
bitvectors, arrays, tuple, fixed-point and floating-point arithmetic (all solvers), and linear integer and real arithmetic (all solvers but Boolector).



Strengths & Weaknesses

- Strengths

- Reach-Safety

- ECA (score: 1113)

- Floats (score: 790)

- Heap (score: 300)

- Product Lines (score: 787)



- Falsification (score: 1916)

- Arithmetic, floating point arithmetic

- User-specified assertions

- the use of invariants increases the number of correct proofs in ESBMC by about 7%

- Weaknesses

- need relational analysis that can keep track of relationship between variables

```
unsigned int s = 1;
int main() {
    while (1) {
        unsigned int input = __VERIFIER_nondet_int();
        if (input > 5) {
            return 0;
        } else if (input == 1 && s == 1) {
            s = 2;
        } else if (input == 2 && s == 2) {
            s = 3;
        } else if (input == 3 && s == 3) {
            s = 4;
        } else if (input == 4 && s == 4) {
            s = 5;
        } else if (input == 5 && s > 5) { // satisfiable
            __VERIFIER_error(); // property violation
        }
    }
}
```

Simplified safe program extracted from SV-COMP 2018

```
unsigned int s = 1;
int main() {
    while (1) {
        unsigned int input = __Verifier_nondet_int();
        if (input > 5) {
            return 0;
        } else if (input == 1 && s == 1) {
            s = 2;
        } else if (input == 2 && s == 2) {
            s = 3;
        } else if (input == 3 && s == 3) {
            s = 4;
        } else if (input == 4 && s == 4) {
            s = 5;
        } else if (input == 5 && s > 5) { // satisfiable
            __Verifier_error(); // property violation
        }
    }
}
```

Program under
verification

Enable *k*-induction
instead of plain BMC

Enable interval
analysis



esbmc main.c --k-induction --interval-analysis

without interval analysis

```
unsigned int s = 1;
int main() {
    while (1) {
        unsigned int
        if (input > 5)
            return 0;
        } else if (input == 5)
            s = 2;
        } else if (input == 4)
            s = 3;
        } else if (input == 3)
            s = 4;
        } else if (input == 2)
            s = 5;
        } else if (input == 5 && s > 5) { // satisfiable
            __VERIFIER_error(); // property violation
        }
    }
}
```

Unwinding loop 1 iteration 49 file svcomp2018.c line 17 function main
Not unwinding loop 1 iteration 50 file svcomp2018.c line 17 function main
Symex completed in: 0.111s (3563 assignments)
Slicing time: 0.008s (removed 2716 assignments)
Generated 1 VCC(s), 1 remaining after simplification (847 assignments)
No solver specified; defaulting to Boolector
Encoding remaining VCC(s) using bit-vector arithmetic
Encoding to solver time: 0.006s
Solving with solver Boolector 3.0.0
Encoding to solver time: 0.006s
Runtime decision procedure: 0.219s
The inductive step is unable to prove the property
Unable to prove or falsify the program, giving up.
VERIFICATION UNKNOWN

with interval analysis

```
unsigned int s = 1;
int main() {
    while (1) {
        unsigned int input = __VERIFIER_nondet_int();
        if (input > 5) {
            return 0;
        } else if (input == 1 && s == 1) {
            s = 2;
        } else if (input == 2 && s == 2) {
            s = 3;
        } else if (input == 3 && s == 3) {
            s = 4;
        } else if (input == 4 && s == 4) {
            s = 5;
        } else if (input == 5 && s > 5) { // satisfiable
            __VERIFIER_error(); // property violation
        }
    }
}
```

1

ASSUME $s \leq 5 \ \&\& \ 1 \leq s$

with interval analysis

```
unsigned int s = 1;
int main() {
    while (1) {
        unsigned int input = __VERIFIER_nondet_int();
        if (input > 5) {
            return 0;
        } else if (input == 1 && s == 1) {
            s = 2;
        } else if (input == 2 && s == 2) {
            s = 3;
        } else if (input == 3 && s == 3) {
            s = 4;
        } else if (input == 4 && s == 4) {
            s = 5;
        } else if (input == 5 && s > 5) { // satisfiable
            __VERIFIER_error(); // property violation
        }
    }
}
```

1

ASSUME $s \leq 5 \ \&\& \ 1 \leq s$

2

ASSUME $s \leq 5 \ \&\& \ 1 \leq s$

with interval analysis

```
unsigned int s = 1;
int main() {
    while (1) {
        unsigned int input = __VERIFIER_nondet_int();
        if (input > 5) {
            return 0;
        } else if (input == 1 && s == 1) {
            s = 2;
        } else if (input == 2 && s == 2) {
            s = 3;
        } else if (input == 3 && s == 3) {
            s = 4;
        } else if (input == 4 && s == 4) {
            s = 5;
        } else if (input == 5 && s > 5) { // satisfiable
            __VERIFIER_error(); // property violation
        }
    }
}
```

The diagram illustrates the flow of interval analysis through the code. Three green circles labeled 1, 2, and 3 are connected by arrows to three green boxes containing ASSUME statements. Circle 1 points to the first ASSUME box, which contains the statement `ASSUME s <= 5 && 1 <= s`. Circle 2 points to the second ASSUME box, which also contains the same statement. Circle 3 points to the third ASSUME box, which contains the statement `ASSUME s <= 5 && 1 <= s && 6 <= input`.

with interval analysis

```
unsigned int s = 1;
int main() {
    while (1) {
        unsigned int input = __VERIFIER_nondet_int();
        if (input > 5) {
            return 0;
        } else if (input == 1 && s == 1) {
            s = 2;
        } else if (input == 2 && s == 2) {
            s = 3;
        } else if (input == 3 && s == 3) {
            s = 4;
        } else if (input == 4 && s == 4) {
            s = 5;
        } else if (input == 5 && s > 5) { // satisfiable
            __VERIFIER_error(); // property violation
        }
    }
}
```

The diagram illustrates the flow of interval analysis assumptions through the code. It shows four points of analysis (labeled 1, 2, 3, and 4) and the resulting assumptions:

- Point 1: ASSUME $s \leq 5 \text{ \&\& } 1 \leq s$
- Point 2: ASSUME $s \leq 5 \text{ \&\& } 1 \leq s$
- Point 3: ASSUME $s \leq 5 \text{ \&\& } 1 \leq s \text{ \&\& } 6 \leq \text{input}$
- Point 4: ASSUME $s == 1 \text{ \&\& } \text{input} == 1$

with interval analysis

```
unsigned int s = 1;
int main() {
    while (1) {
        unsigned int input = __VERIFIER_nondet_int();
        if (input > 5) {
            return 0;
        } else if (input == 1 && s == 1) {
            s = 2;
        } else if (input == 2 && s == 2) {
            s = 3;
        } else if (input == 3 && s == 3) {
            s = 4;
        } else if (input == 4 && s == 4) {
            s = 5;
        } else if (input == 5 && s > 5) { // satisfiable
            __VERIFIER_error(); // property violation
        }
    }
}
```

The diagram illustrates the flow of interval analysis assumptions through the code. Five green circles, labeled 1 through 5, are connected by arrows to five corresponding assumption boxes above the code. Circle 1 points to the first assumption box. Circle 2 points to the second. Circle 3 points to the third. Circle 4 points to the fourth. Circle 5 points to the fifth.

- ASSUME $s \leq 5 \ \&\& 1 \leq s$
- ASSUME $s \leq 5 \ \&\& 1 \leq s$
- ASSUME $s \leq 5 \ \&\& 1 \leq s \ \&\& 6 \leq \text{input}$
- ASSUME $s == 1 \ \&\& \text{input} == 1$
- ASSUME $s == 2 \ \&\& \text{input} == 2$

with interval analysis

```
unsigned int s = 1;
int main() {
    while (1) {
        unsigned int input = __VERIFIER_nondet_int();
        if (input > 5) {
            return 0;
        } else if (input == 1 && s == 1) {
            s = 2;
        } else if (input == 2 && s == 2) {
            s = 3;
        } else if (input == 3 && s == 3) {
            s = 4;
        } else if (input == 4 && s == 4) {
            s = 5;
        } else if (input == 5 && s > 5) { // satisfiable
            __VERIFIER_error(); // property violation
        }
    }
}
```

The diagram illustrates the flow of interval analysis assumptions through the code. Six green circles, labeled 1 through 6, are connected by arrows to specific points in the code, each associated with a green rectangular box containing an 'ASSUME' statement:

- Circle 1: ASSUME $s \leq 5 \ \&\& 1 \leq s$
- Circle 2: ASSUME $s \leq 5 \ \&\& 1 \leq s$
- Circle 3: ASSUME $s \leq 5 \ \&\& 1 \leq s \ \&\& 6 \leq \text{input}$
- Circle 4: ASSUME $s == 1 \ \&\& \text{input} == 1$
- Circle 5: ASSUME $s == 2 \ \&\& \text{input} == 2$
- Circle 6: ASSUME $s == 3 \ \&\& \text{input} == 3$

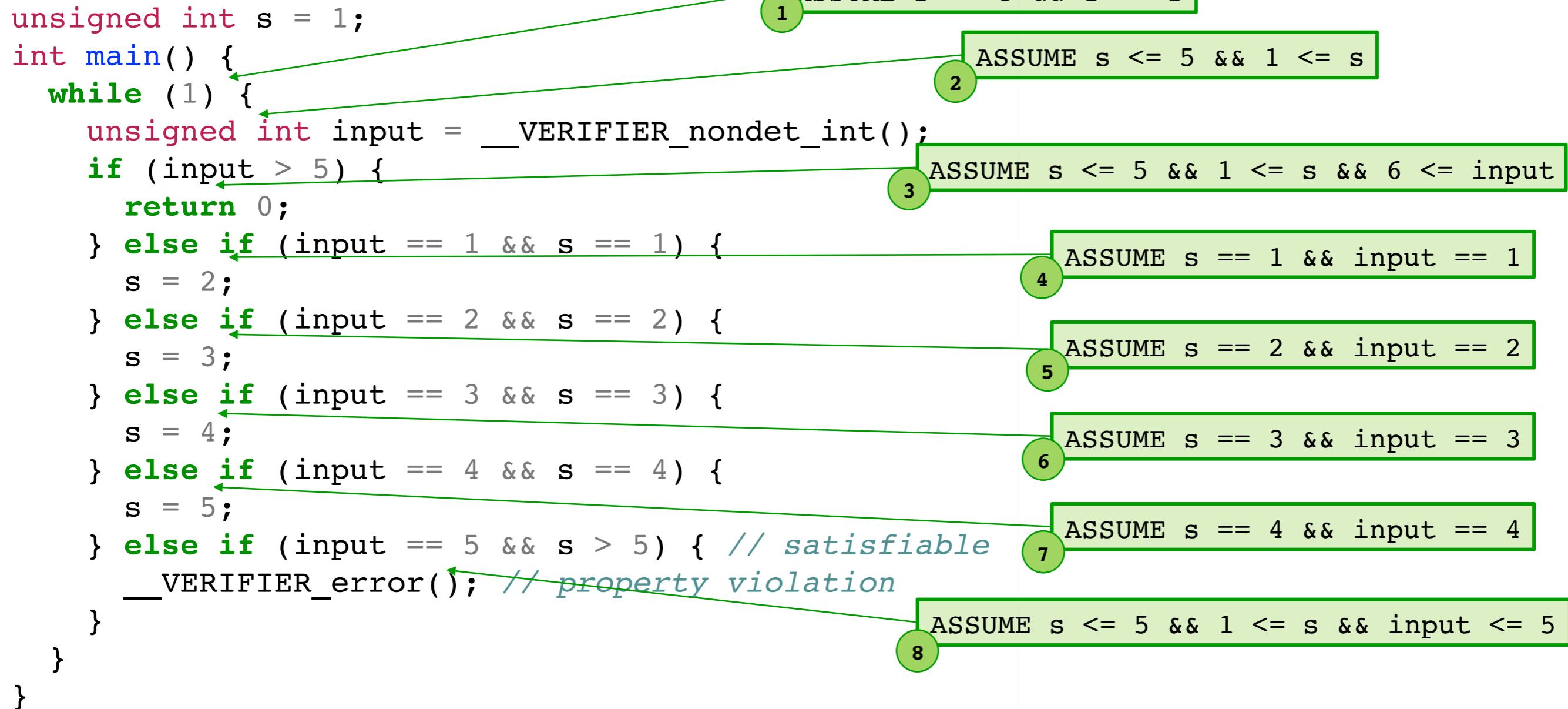
with interval analysis

```
unsigned int s = 1;
int main() {
    while (1) {
        unsigned int input = __VERIFIER_nondet_int();
        if (input > 5) {
            return 0;
        } else if (input == 1 && s == 1) {
            s = 2;
        } else if (input == 2 && s == 2) {
            s = 3;
        } else if (input == 3 && s == 3) {
            s = 4;
        } else if (input == 4 && s == 4) {
            s = 5;
        } else if (input == 5 && s > 5) { // satisfiable
            __VERIFIER_error(); // property violation
        }
    }
}
```

The diagram illustrates the verification process using interval analysis. It shows the state of variable s and input $input$ at seven different points (labeled 1 through 7) during the execution of the `main` function.

- Point 1:** $s \leq 5 \text{ \&\& } 1 \leq s$
- Point 2:** $s \leq 5 \text{ \&\& } 1 \leq s$
- Point 3:** $s \leq 5 \text{ \&\& } 1 \leq s \text{ \&\& } 6 \leq \text{input}$
- Point 4:** $s == 1 \text{ \&\& } \text{input} == 1$
- Point 5:** $s == 2 \text{ \&\& } \text{input} == 2$
- Point 6:** $s == 3 \text{ \&\& } \text{input} == 3$
- Point 7:** $s == 4 \text{ \&\& } \text{input} == 4$

Arrows indicate the flow from Point 1 to Point 2, Point 2 to Point 3, Point 3 to Point 4, Point 3 to Point 5, Point 3 to Point 6, and Point 3 to Point 7. The final part of the code, after Point 7, is annotated with *// satisfiable* and *// property violation*.

with interval analysis

with interval analysis

```
unsigned int s = 1;
```

```
int main() {
    while (1) {
```

```
        unsigned int input = __VERIFIER_nondet_int();
```

```
        if (input > 5) {
```

```
            return
```

```
} else i
```

```
    s = 2;
```

```
} else i
```

```
    s = 3;
```

```
} else i
```

```
    s = 4;
```

```
} else i
```

```
    s = 5;
```

```
} else i
```

```
    __VERI
```

```
}
```

```
}
```

1

2

3

ASSUME s <= 5 && 1 <= s

ASSUME s <= 5 && 1 <= s

ASSUME s <= 5 && 1 <= s && 6 <= input

```
*** Checking inductive step
Starting Bounded Model Checking
Unwinding loop 1 iteration 1 file svcomp2018.c line 17 function main
Not unwinding loop 1 iteration 2 file svcomp2018.c line 17 function main
Symex completed in: 0.001s (82 assignments)
Slicing time: 0.000s (removed 22 assignments)
Generated 1 VCC(s), 1 remaining after simplification (60 assignments)
No solver specified; defaulting to Boolector
Encoding remaining VCC(s) using bit-vector arithmetic
Encoding to solver time: 0.000s
Solving with solver Boolector 3.0.0
Encoding to solver time: 0.000s
Runtime decision procedure: 0.001s
BMC program time: 0.003s
```

ut == 1

ut == 2

ut == 3

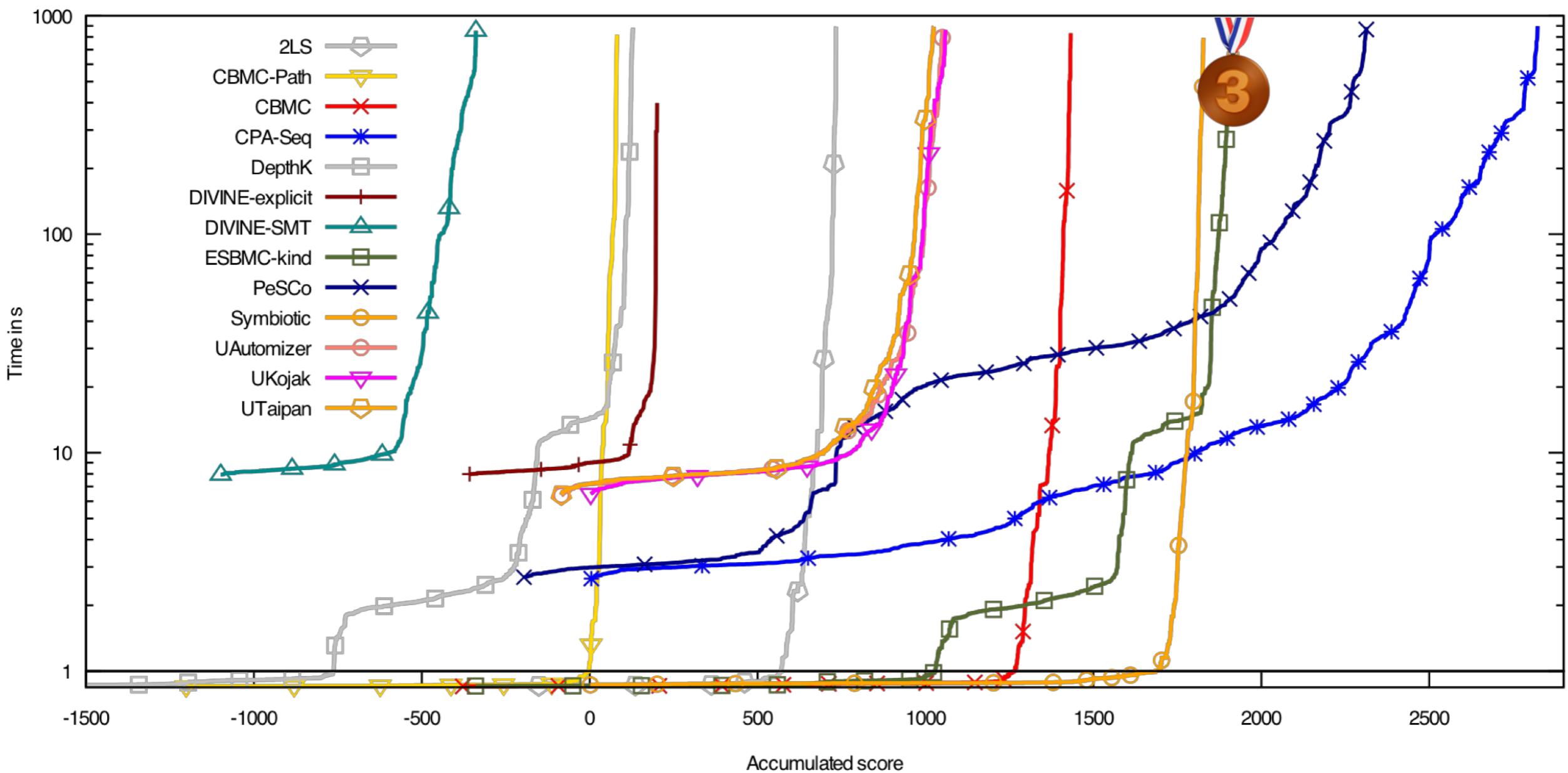
ut == 4

input <= 5

VERIFICATION SUCCESSFUL

Solution found by the inductive step (k = 2)

Falsification



Thank you!

More information available at <http://esbmc.org/>

