

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

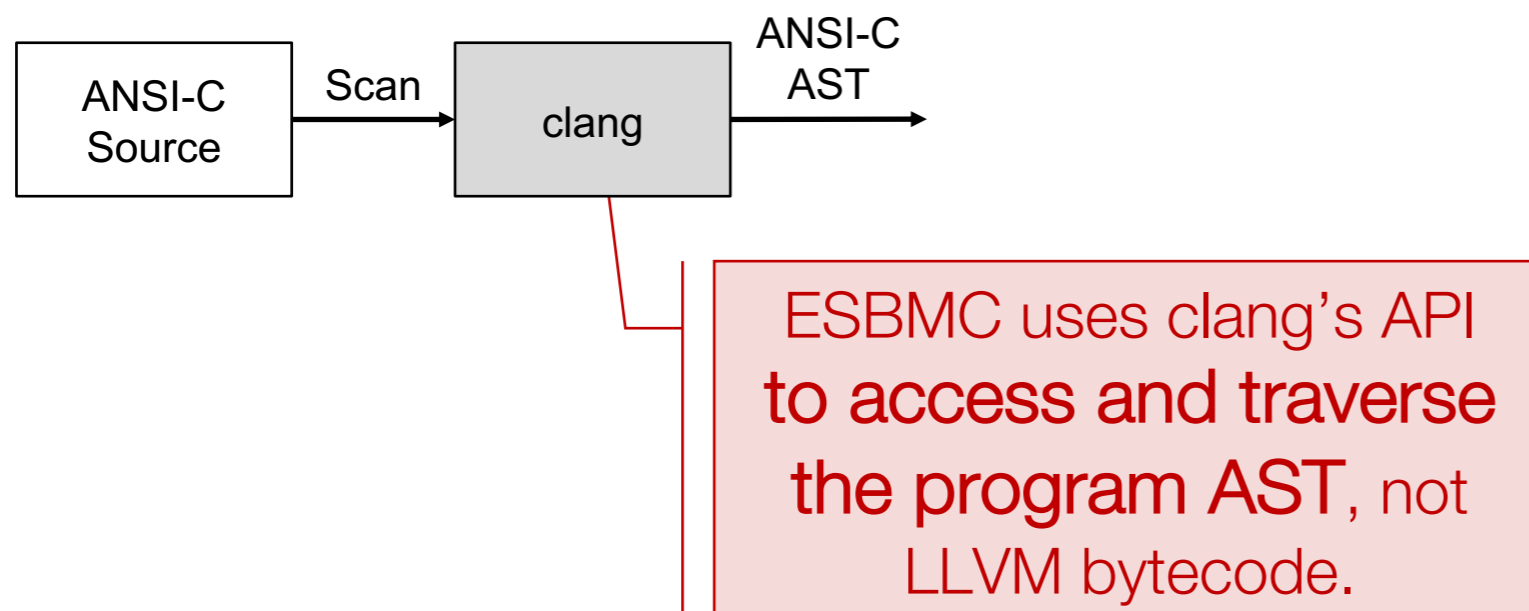
ESBMC Architecture

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

ANSI-C
Source

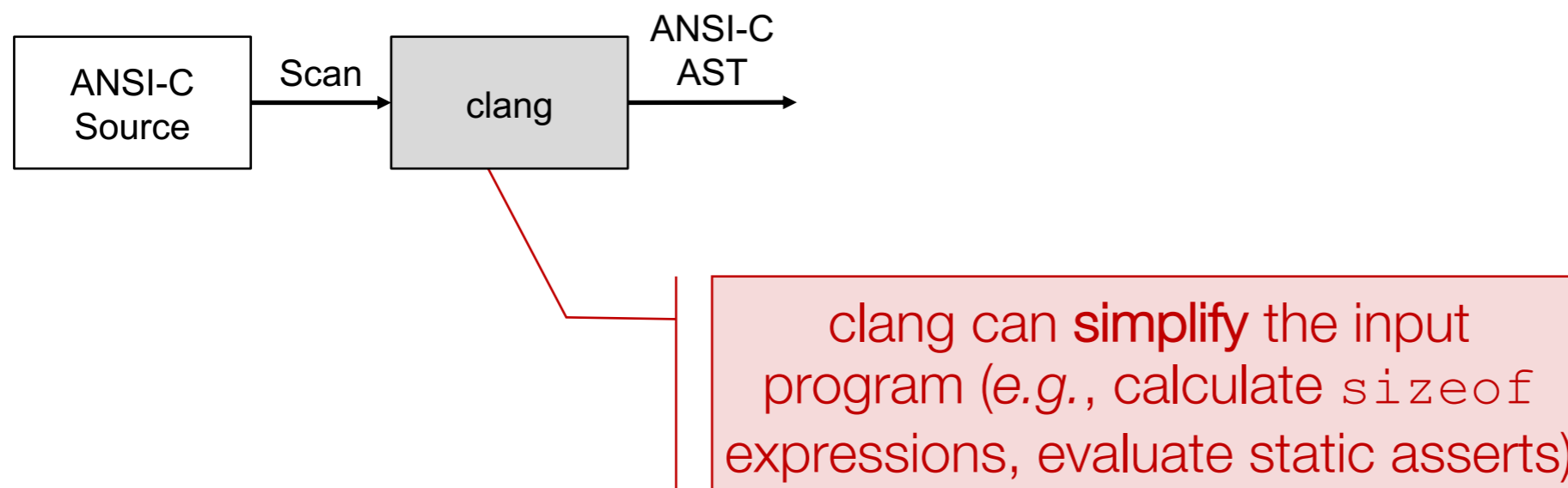
ESBMC Architecture

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



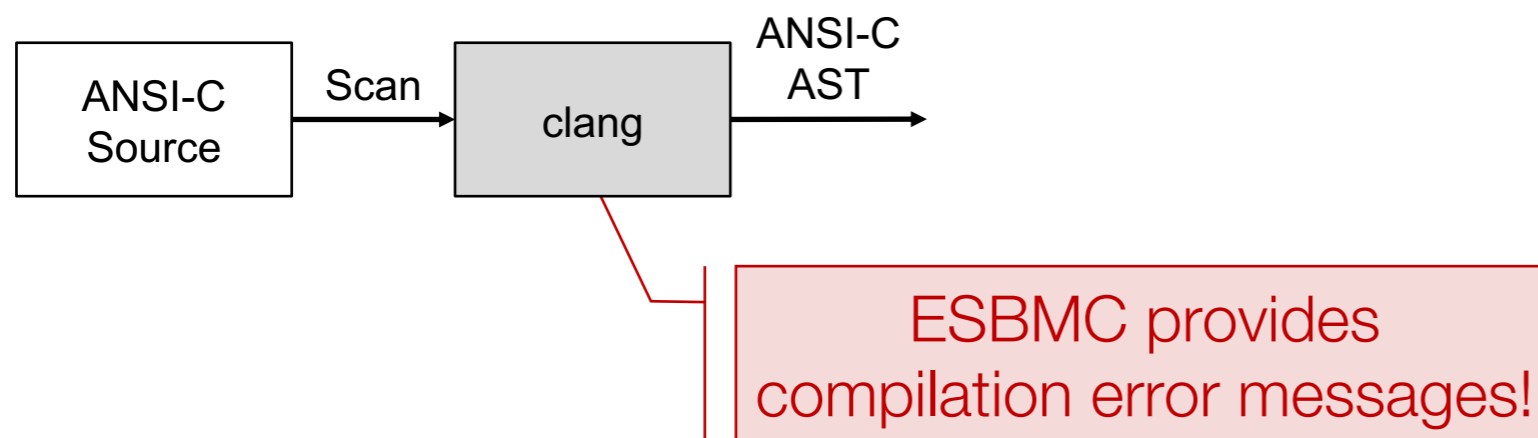
ESBMC Architecture

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



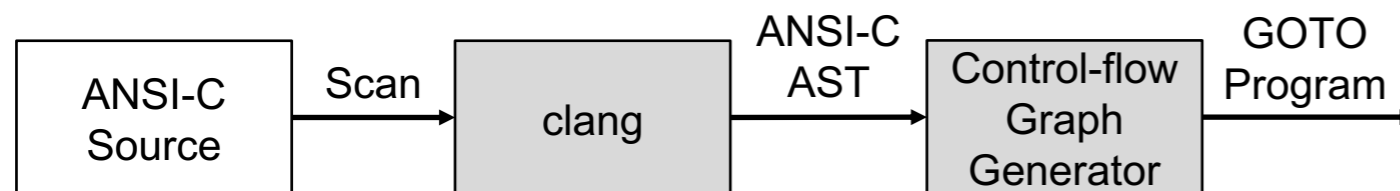
ESBMC Architecture

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



ESBMC Architecture

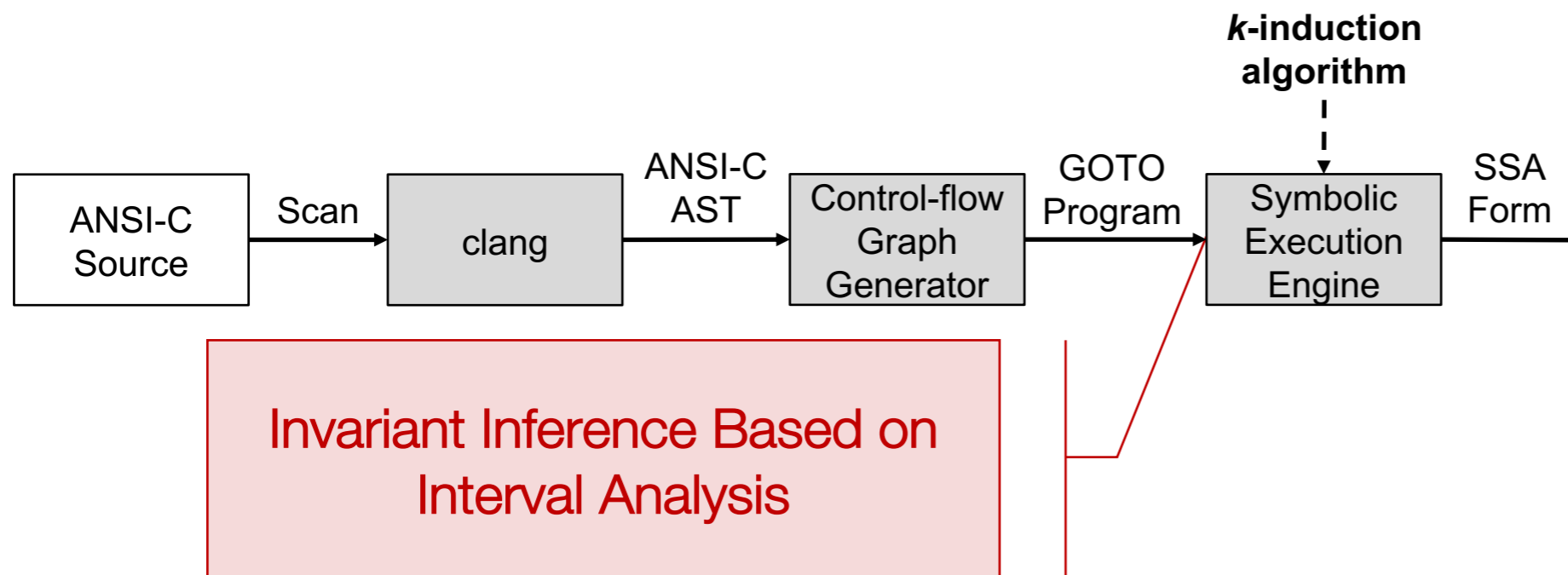
- The CFG generator takes the program AST and transforms it into an equivalent GOTO program
 - only of assignments, conditional and unconditional branches, assumes, and assertions.



ESBMC Architecture

- ESBMC perform a static analysis prior to loop unwinding and (over-)estimate the range that a variable can assume

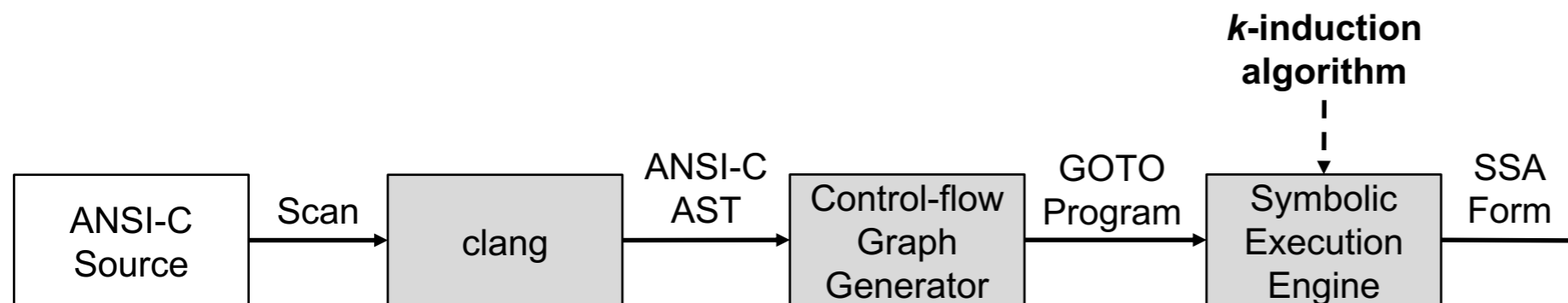
“rectangular” invariant generation based on interval analysis (e.g., $a \leq x \leq b$)



- 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](#)

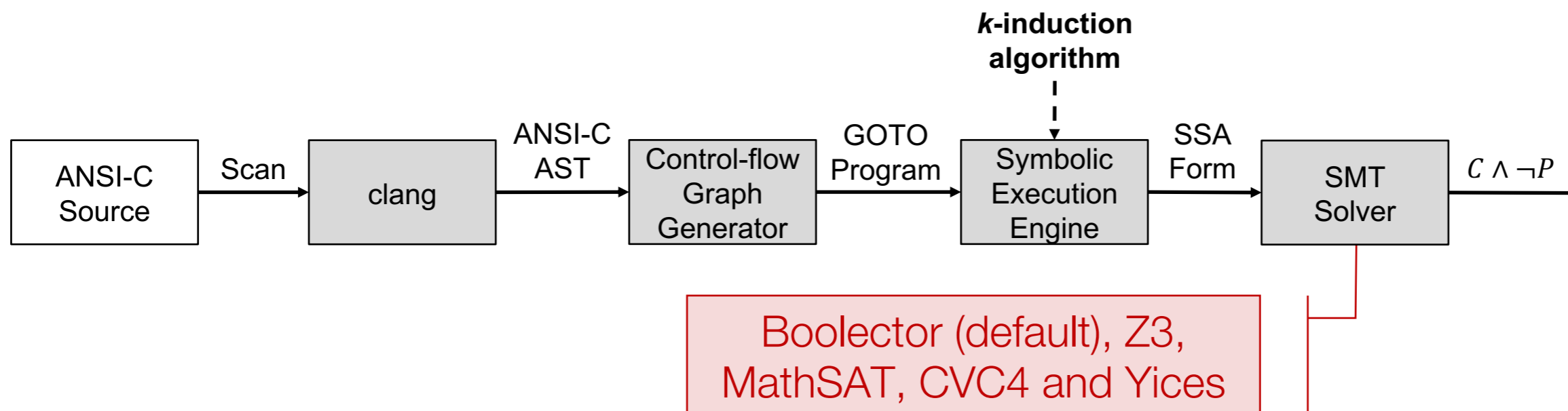
[Authors and affiliations](#)

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

ESBMC Architecture

- The back-end is highly configurable and allows the encoding of quantifier-free formulas

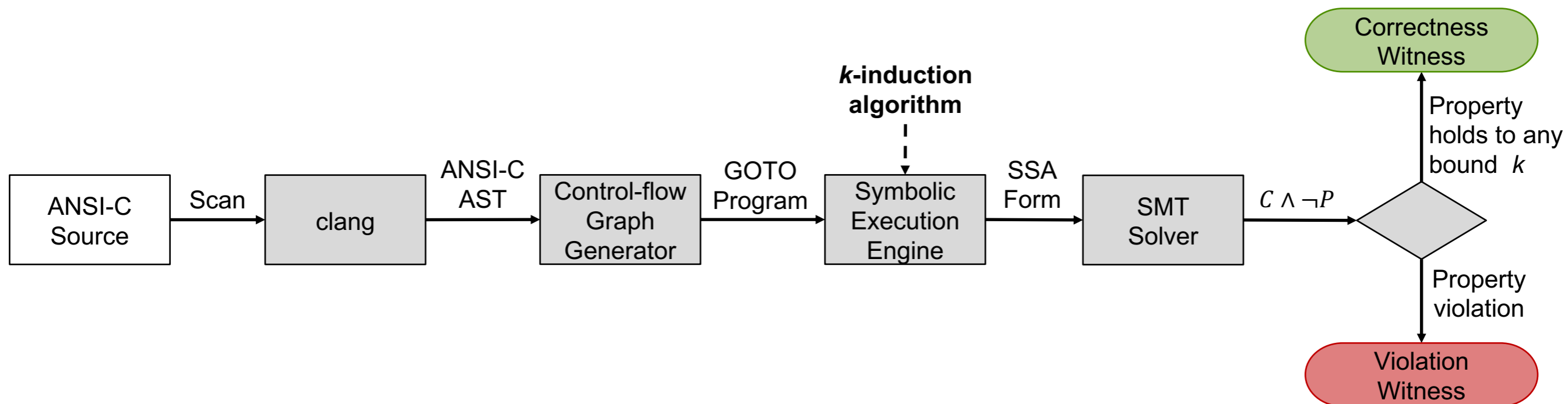
bitvectors, arrays, tuple, fixed-point and floating-point arithmetic (all solvers), and linear integer and real arithmetic (all solvers but Boolector).



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) { // unsatisfiable
      __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) { // unsatisfiable
      __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
      s = 2;
    } else if (input
      s = 3;
    } else if (input
      s = 4;
    } else if (input
      s = 5;
    } else if (input == 5 && s > 5) { // unsatisfiable
      __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) { // unsatisfiable
      __VERIFIER_error(); // property violation
    }
  }
}
```

1

ASSUME s <= 5 && 1 <= 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) { // unsatisfiable
      __VERIFIER_error(); // property violation
    }
  }
}
```

1 ASSUME s <= 5 && 1 <= s

2 ASSUME s <= 5 && 1 <= 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) { // unsatisfiable
      __VERIFIER_error(); // property violation
    }
  }
}

```

Annotations illustrating interval analysis assumptions:

- 1: ASSUME `s <= 5 && 1 <= s` (at the start of the `while` loop)
- 2: ASSUME `s <= 5 && 1 <= s` (at the start of the `if` statement)
- 3: ASSUME `s <= 5 && 1 <= s && 6 <= input` (at the start of the `return 0;` branch)

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) { // unsatisfiable
      __VERIFIER_error(); // property violation
    }
  }
}

```

Annotations and flow control:

- 1: ASSUME s <= 5 && 1 <= s (at the start of the while loop)
- 2: ASSUME s <= 5 && 1 <= s (at the start of the if statement)
- 3: ASSUME s <= 5 && 1 <= s && 6 <= input (at the start of the return 0; statement)
- 4: ASSUME s == 1 && input == 1 (at the start of the first else if branch)

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) { // unsatisfiable
      __VERIFIER_error(); // property violation
    }
  }
}

```

Diagram illustrating the application of interval analysis to the provided code. The analysis tracks the state of variables `s` and `input` through various branches of the program. The assumptions are summarized in the following table:

Assumption ID	Assumption Statement
1	ASSUME <code>s <= 5 && 1 <= s</code>
2	ASSUME <code>s <= 5 && 1 <= s</code>
3	ASSUME <code>s <= 5 && 1 <= s && 6 <= input</code>
4	ASSUME <code>s == 1 && input == 1</code>
5	ASSUME <code>s == 2 && input == 2</code>

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) { // unsatisfiable
      __VERIFIER_error(); // property violation
    }
  }
}

```

Diagram illustrating the application of interval analysis to the provided C code. The code is annotated with six ASSUME statements (1-6) that track the state of variables `s` and `input` during execution. Green arrows indicate the flow of control from the code to the corresponding ASSUME statement.

- 1: ASSUME `s <= 5 && 1 <= s` (at the start of the `while` loop)
- 2: ASSUME `s <= 5 && 1 <= s` (at the start of the `if` statement)
- 3: ASSUME `s <= 5 && 1 <= s && 6 <= input` (at the start of the `return 0;` branch)
- 4: ASSUME `s == 1 && input == 1` (at the start of the `else if (input == 1 && s == 1)` branch)
- 5: ASSUME `s == 2 && input == 2` (at the start of the `else if (input == 2 && s == 2)` branch)
- 6: ASSUME `s == 3 && input == 3` (at the start of the `else if (input == 3 && s == 3)` branch)

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) { // unsatisfiable
      __VERIFIER_error(); // property violation
    }
  }
}

```

Annotations illustrating interval analysis assumptions:

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

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) { // unsatisfiable
      __VERIFIER_error(); // property violation
    }
  }
}

```

Diagram illustrating the application of interval analysis to the provided C code. The code is annotated with eight ASSUME statements (1-8) that track the state of variables `s` and `input` at various points in the execution flow.

- 1:** ASSUME `s <= 5 && 1 <= s` (Initial state)
- 2:** ASSUME `s <= 5 && 1 <= s` (Before the `while` loop)
- 3:** ASSUME `s <= 5 && 1 <= s && 6 <= input` (Before the `if` statement)
- 4:** ASSUME `s == 1 && input == 1` (Inside the `else if` branch for `input == 1`)
- 5:** ASSUME `s == 2 && input == 2` (Inside the `else if` branch for `input == 2`)
- 6:** ASSUME `s == 3 && input == 3` (Inside the `else if` branch for `input == 3`)
- 7:** ASSUME `s == 4 && input == 4` (Inside the `else if` branch for `input == 4`)
- 8:** ASSUME `s <= 5 && 1 <= s && input <= 5` (Before the `__VERIFIER_error()` call)

with interval analysis

```

unsigned int s = 1;
int main() {
  while (1) {
    unsigned int input = __VERIFIER_nondet_int();
    if (input > 5) {
      return;
    } else if (input == 1) {
      s = 2;
    } else if (input == 2) {
      s = 3;
    } else if (input == 3) {
      s = 4;
    } else if (input == 4) {
      s = 5;
    } else if (input == 5) {
      __VERIFIER_assert(input <= 5);
    }
  }
}

```

1 ASSUME s <= 5 && 1 <= s

2 ASSUME s <= 5 && 1 <= s

3 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

```

VERIFICATION SUCCESSFUL

Solution found by the inductive step (k = 2)

input == 1

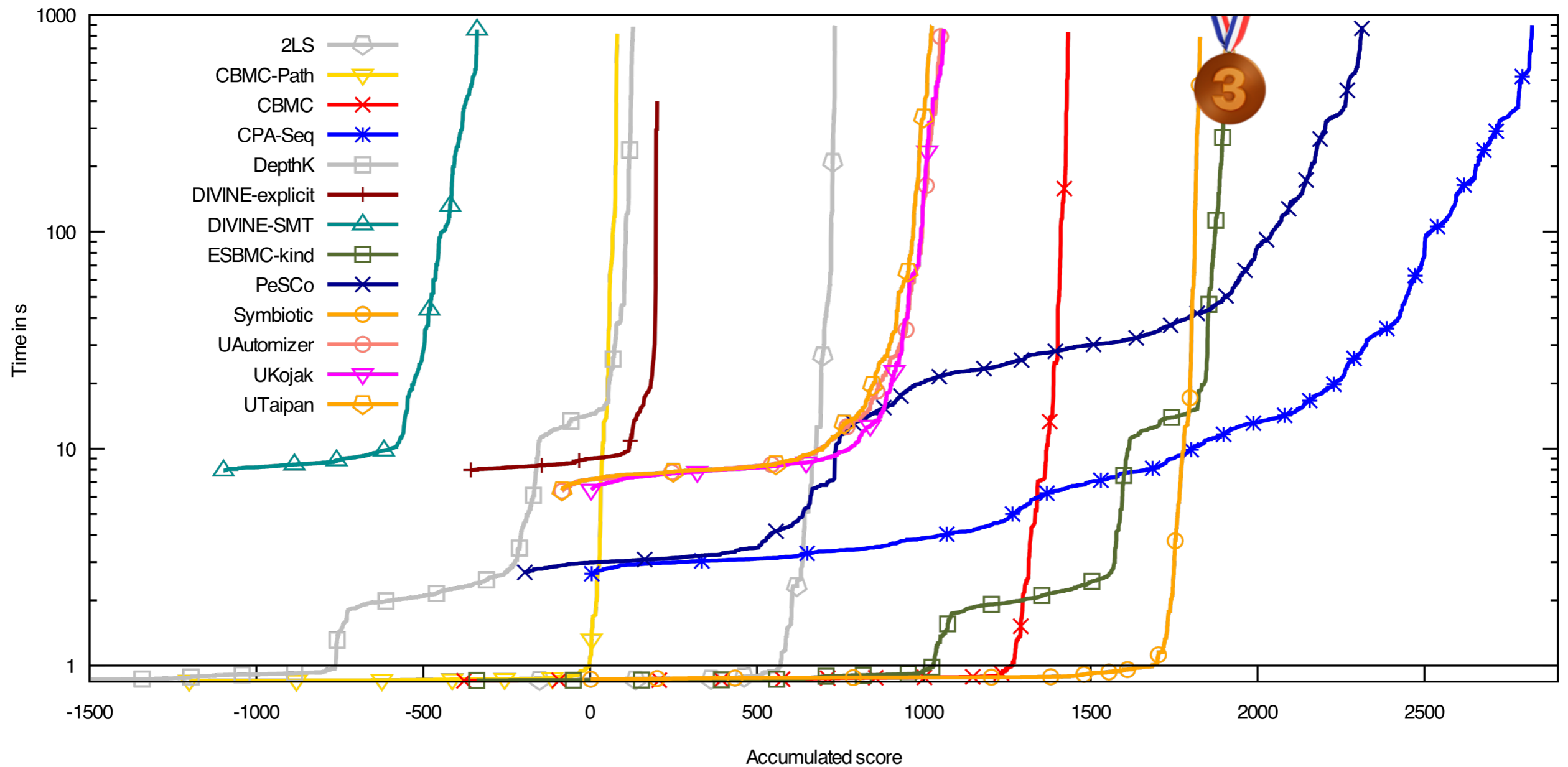
input == 2

input == 3

input == 4

input <= 5

Falsification



Thank you!

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

