Exploiting the SAT Revolution for Automated Software Verification

Lucas Cordeiro
Department of Computer Science
lucas.cordeiro@manchester.ac.uk
https://ssvlab.github.io/lucasccordeiro/

Memory Safety Vulnerabilities

Memory errors in **low-level systems software** written in **unsafe programming languages** such as C or C++ represent one of the main problems in **computer security**

- The top ten vulnerabilities in CWE include **four types of memory errors** (out of bounds and use after free)
- Microsoft reports that around **70%** of all security updates in their products address **memory issues**
- Google reports a similar number regarding bugs in the Chrome Browser

Research Questions

Given a **computer program** and a **specification**, can we automatically **verify** that the **program performs as specified**?

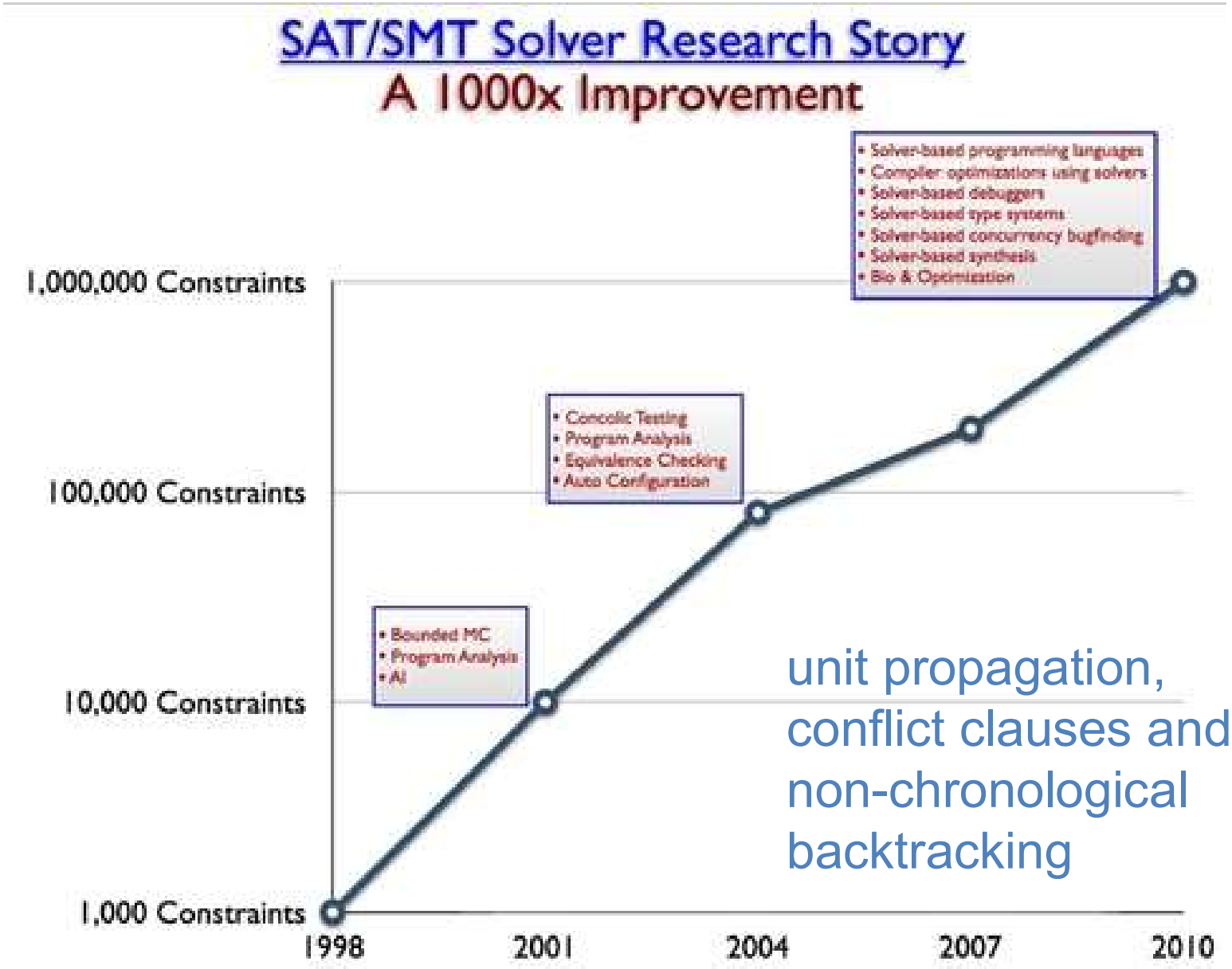
Can we leverage **program analysis/synthesis** to discover more **software vulnerabilities** than existing state-of-the-art approaches?

Objective of this talk

Discuss the past, present, and future of **software model checking** based on **SAT/SMT solving**

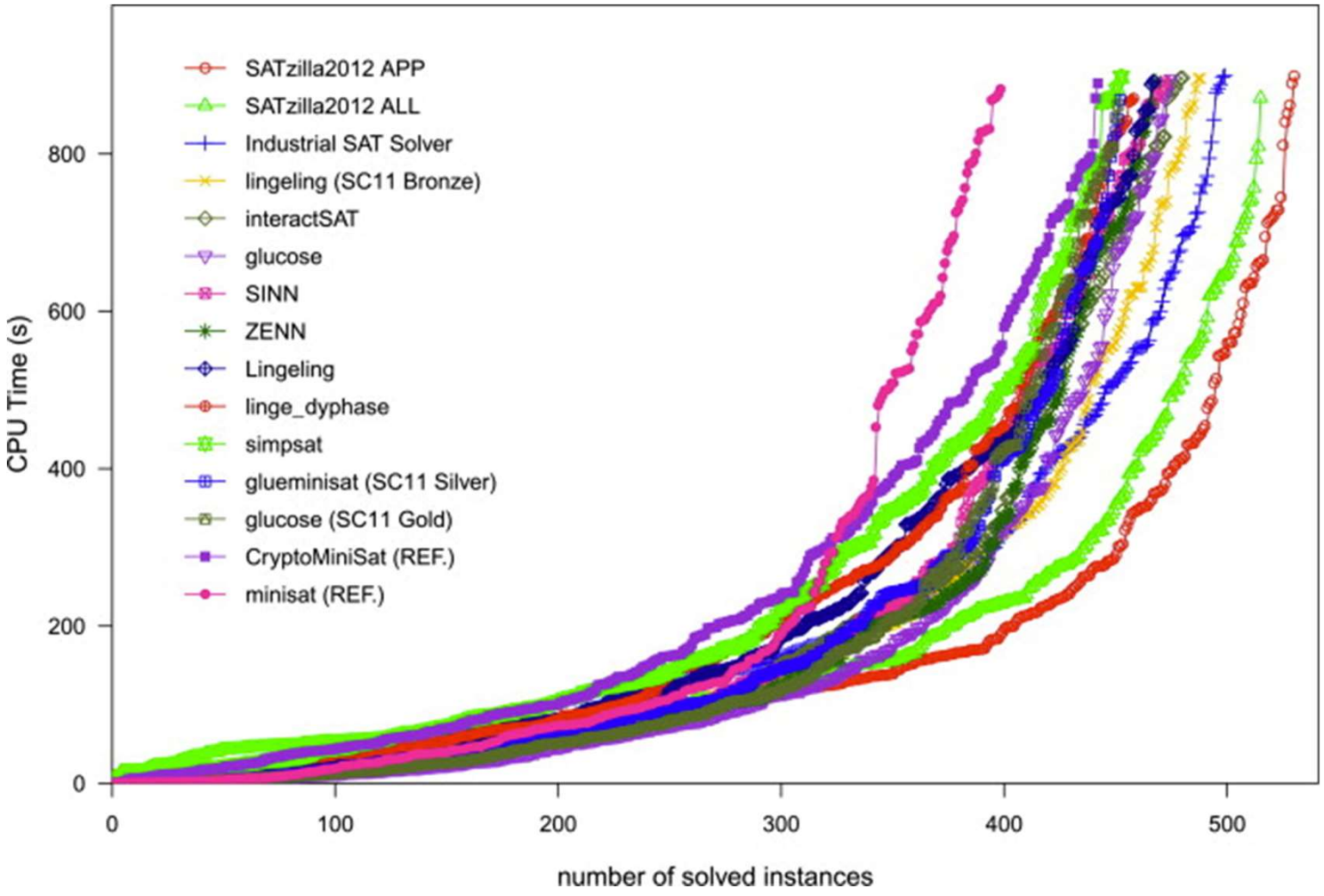
- BMC analyzes bounded program runs (achieving decidability)
- SAT solvers handle formulas with millions of variables
- There exist better encodings using word-level theories
- Invariant inference and induction help verify more programs than plain BMC

SAT solving as enabling technology



Kroening, D., Strichman, O., Decision Procedures - An Algorithmic Point of View, Second Edition, Springer.

SAT Competition



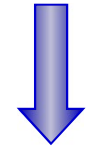
http://www.satcompetition.org/

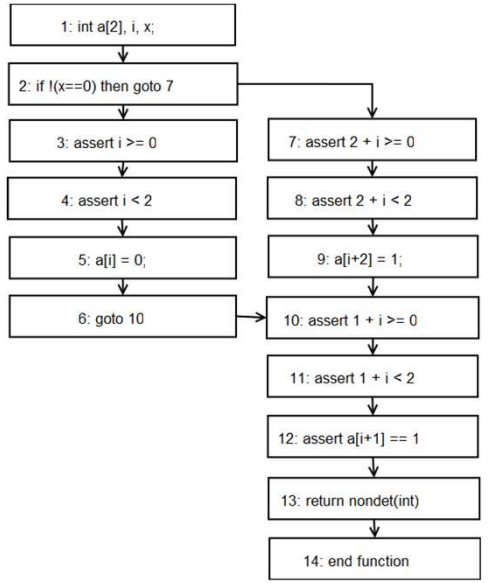
Software BMC

- program modelled as transition system
  - *state*: *pc* and program variables
  - derived from control-flow graph

```
int getPassword() {
  char buf[2];
  gets(buf);
  return strcmp(buf, "ML");
}
```

```
void main(){
  int x=getPassword();
  if(x){
    printf("Access Denied\n");
    exit(0);
  }
  printf("Access Granted\n");
}
```



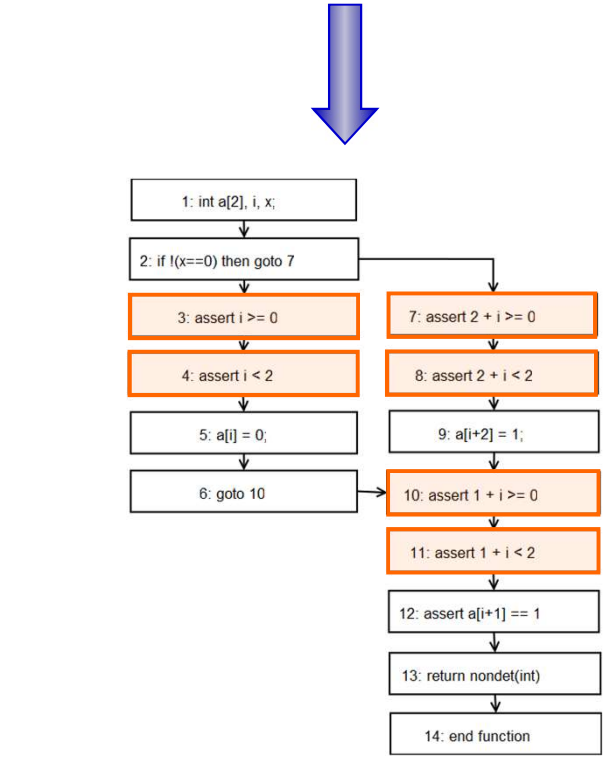


Software BMC

- program modelled as transition system
  - *state*: *pc* and program variables
  - derived from control-flow graph
  - added safety properties as extra nodes

```
int getPassword() {
  char buf[2];
  gets(buf);
  return strcmp(buf, "ML");
}
```

```
void main(){
  int x=getPassword();
  if(x){
    printf("Access Denied\n");
    exit(0);
  }
  printf("Access Granted\n");
}
```

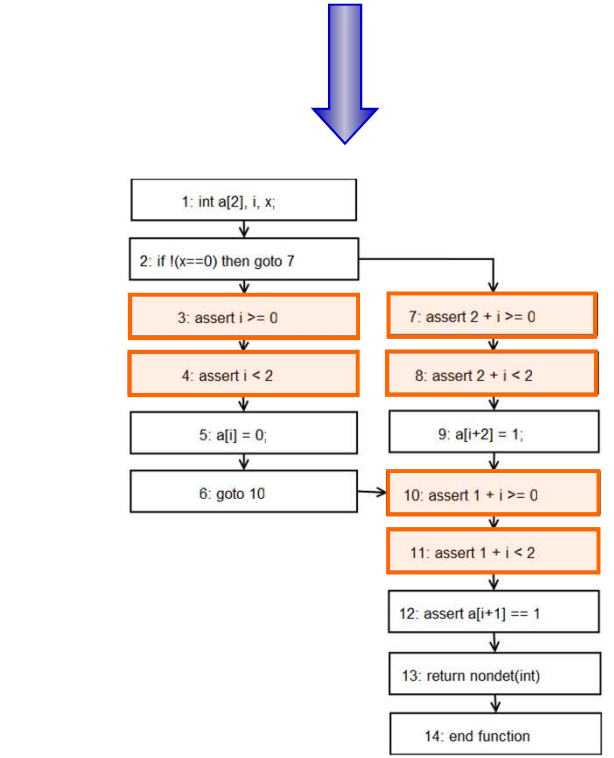


Software BMC

- program modelled as transition system
  - *state*: *pc* and program variables
  - derived from control-flow graph
  - added safety properties as extra nodes
- program unfolded up to given bounds

```
int getPassword() {
  char buf[2];
  gets(buf);
  return strcmp(buf, "ML");
}
```

```
void main(){
  int x=getPassword();
  if(x){
    printf("Access Denied\n");
    exit(0);
  }
  printf("Access Granted\n");
}
```



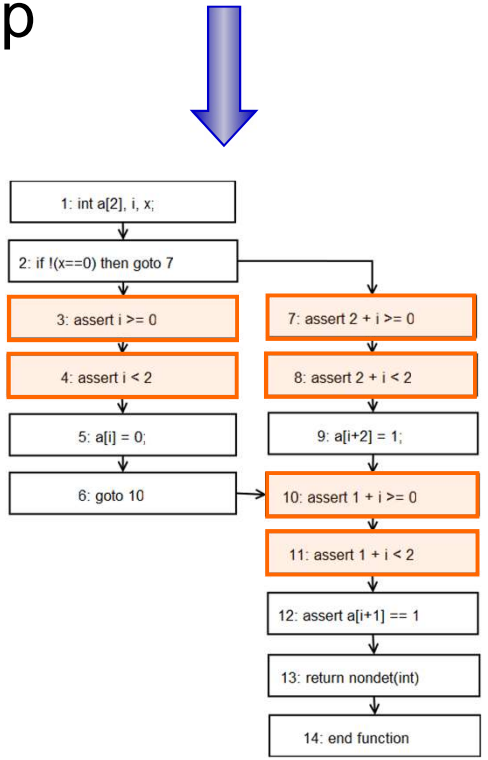
Software BMC

- program modelled as transition system
  - *state*: *pc* and program variables
  - derived from control-flow graph
  - added safety properties as extra nodes
- program unfolded up to given bounds
- unfolded program optimized to reduce blow-up
  - constant propagation
  - forward substitutions
  - unreachable code

} crucial

```
int getPassword() {
  char buf[2];
  gets(buf);
  return strcmp(buf, "ML");
}
```

```
void main(){
  int x=getPassword();
  if(x){
    printf("Access Denied\n");
    exit(0);
  }
  printf("Access Granted\n");
}
```



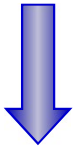
Software BMC

- program modelled as transition system
  - *state*: *pc* and program variables
  - derived from control-flow graph
  - added safety properties as extra nodes
- program unfolded up to given bounds
- unfolded program optimized to reduce blow-up
  - constant propagation
  - forward substitutions
  - unreachable code

  } crucial
- front-end converts unrolled and **optimized program into SSA**

```
int getPassword() {
  char buf[2];
  gets(buf);
  return strcmp(buf, "ML");
}
```

```
void main(){
  int x=getPassword();
  if(x){
    printf("Access Denied\n");
    exit(0);
  }
  printf("Access Granted\n");
}
```



$g_1 = x_1 == 0$
$a_1 = a_0$ WITH $[i_0:=0]$
$a_2 = a_0$
$a_3 = a_2$ WITH $[2+i_0:=1]$
$a_4 = g_1 \ ? \ a_1 : a_3$
$t_1 = a_4 [1+i_0] == 1$

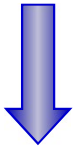
Software BMC

- program modelled as transition system
  - *state*: *pc* and program variables
  - derived from control-flow graph
  - added safety properties as extra nodes
- program unfolded up to given bounds
- unfolded program optimized to reduce blow-up
  - constant propagation
  - forward substitutions
  - unreachable code

  } crucial
- front-end converts unrolled and **optimized program into SSA**
- extraction of *constraints C* and *properties P*

```
int getPassword() {
  char buf[2];
  gets(buf);
  return strcmp(buf, "ML");
}
```

```
void main(){
  int x=getPassword();
  if(x){
    printf("Access Denied\n");
    exit(0);
  }
  printf("Access Granted\n");
}
```



$$C := \begin{bmatrix} g_1 := (x_1 = 0) \\ \wedge\, a_1 := store(a_0, i_0, 0) \\ \wedge\, a_2 := a_0 \\ \wedge\, a_3 := store(a_2, 2 + i_0, 1) \\ \wedge\, a_4 := ite(g_1, a_1, a_3) \end{bmatrix}$$

$$P := \begin{bmatrix} i_0 \geq 0 \wedge i_0 < 2 \\ \wedge\, 2 + i_0 \geq 0 \wedge 2 + i_0 < 2 \\ \wedge\, 1 + i_0 \geq 0 \wedge 1 + i_0 < 2 \\ \wedge\, select(a_4, i_0 + 1) = 1 \end{bmatrix}$$

Software BMC

```
int getPassword() {
  char buf[2];
  gets(buf);
  return strcmp(buf, "ML");
}

void main(){
  int x=getPassword();
  if(x){
    printf("Access Denied\n");
    exit(0);
  }
  printf("Access Granted\n");
}
```

- program modelled as transition system
  - *state*: *pc* and program variables
  - derived from control-flow graph
  - added safety properties as extra nodes
- program unfolded up to given bounds
- unfolded program optimized to reduce blow-up
  - constant propagation
  - forward substitutions
  - unreachable code

  } crucial
- front-end converts unrolled and **optimized program into SSA**
- extraction of *constraints C* and *properties P*
  - specific to selected SMT solver, uses theories

$$C := \begin{bmatrix} g_1 := (x_1 = 0) \\ \wedge\, a_1 := store(a_0, i_0, 0) \\ \wedge\, a_2 := a_0 \\ \wedge\, a_3 := store(a_2, 2 + i_0, 1) \\ \wedge\, a_4 := ite(g_1, a_1, a_3) \end{bmatrix}$$

$$P := \begin{bmatrix} i_0 \geq 0 \wedge i_0 < 2 \\ \wedge\, 2 + i_0 \geq 0 \wedge 2 + i_0 < 2 \\ \wedge\, 1 + i_0 \geq 0 \wedge 1 + i_0 < 2 \\ \wedge\, select(a_4, i_0 + 1) = 1 \end{bmatrix}$$

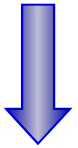
Software BMC

```
int getPassword() {
  char buf[2];
  gets(buf);
  return strcmp(buf, "ML");
}

void main(){
  int x=getPassword();
  if(x){
    printf("Access Denied\n");
    exit(0);
  }
  printf("Access Granted\n");
}
```

- program modelled as transition system
  - *state*: *pc* and program variables
  - derived from control-flow graph
  - added safety properties as extra nodes
- program unfolded up to given bounds
- unfolded program optimized to reduce blow-up
  - constant propagation
  - forward substitutions
  - unreachable code

  } crucial



$$C := \begin{bmatrix} g_1 := (x_1 = 0) \\ \wedge\, a_1 := store(a_0, i_0, 0) \\ \wedge\, a_2 := a_0 \\ \wedge\, a_3 := store(a_2, 2 + i_0, 1) \\ \wedge\, a_4 := ite(g_1, a_1, a_3) \end{bmatrix}$$

- front-end converts unrolled and **optimized program into SSA**
- extraction of *constraints C* and *properties P*
  - specific to selected SMT solver, uses theories

$$P := \begin{bmatrix} i_0 \geq 0 \wedge i_0 < 2 \\ \wedge\, 2 + i_0 \geq 0 \wedge 2 + i_0 < 2 \\ \wedge\, 1 + i_0 \geq 0 \wedge 1 + i_0 < 2 \\ \wedge\, select(a_4, i_0 + 1) = 1 \end{bmatrix}$$

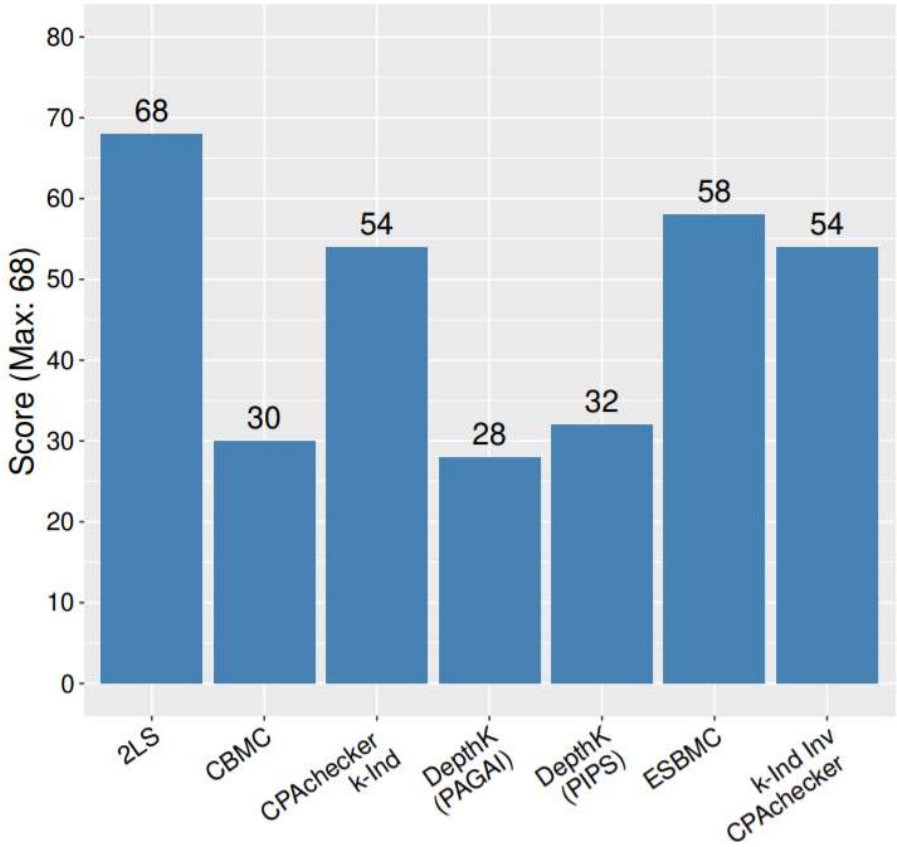
- satisfiability check of $C \wedge \neg P$

Cordeiro et al.: SMT-Based Bounded Model Checking for Embedded ANSI-C Software. IEEE TSE, 2012

Embedded Software Verification

- **Powerstone:** automotive-control and fax applications
- **Real-Time SNU:** matrix handling and signal processing, cyclic-redundancy check, Fourier transform, and JPEG encoding
- **WCET:** a set of programs for executing worst-case time analysis

34 tasks; 900s, 15GB
ESBMC achieved the 2nd place



Alhawi et al.: Verification and refutation of C programs based on k-induction and invariant inference. STTT, 2021

Difficulties in proving the correctness of programs with loops in BMC

- BMC techniques can falsify properties up to a given depth *k*
  - prove correctness if an upper bound of *k* is known
    - » BMC tools typically fail to verify programs with bounded or unbounded loops

```
#include <assert.h>
int main() {
  int x = 1, y = 0;
  while (y < 1000
    && nondet_int()) {
      x = x + y;
      y = y + 1;
  }
  assert(x >= y);
  return 0;
}
```

Does this program terminate?

BMC of Software Using Interval Methods via Contractors

1) Analyze intervals and properties
   - Static Analysis
2) Convert the problem into a CSP
   - Variables, Domains and Constraints
3) Apply contractor to CSP
   - Forward-Backward Contractor
4) Apply reduced intervals back to the program

```
1 unsigned int x=nondet_uint();
2 unsigned int y=nondet_uint();
3 __ESBMC_assume(x >= 20 && x <= 30);
4 __ESBMC_assume(y <= 30);
5 assert(x >= y);
```

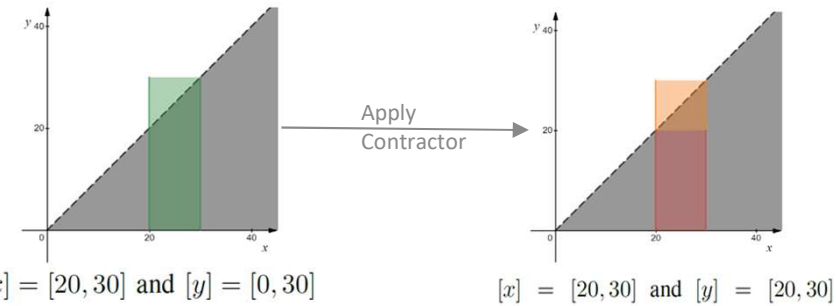
```
__ESBMC_assume(y <= 30 && y >= 20);
```

This **assumption** prunes our search space to the **orange area**

```
1 unsigned int x=nondet_uint();
2 unsigned int y=nondet_uint();
3 __ESBMC_assume(x >= 20 && x <= 30);
4 __ESBMC_assume(y <= 30);
5 assert(x >= y);
```

Domain: $[x] = [20, 30]$ and $[y] = [0, 30]$

Constraint: $y - x \leq 0$



$f(x) > 0$	$I = [0, \infty)$	
$f(x) = y - x$	$[f(x)_1] = I \cap [y_0] - [x_0]$	Forward-step
$x = y - f(x)$	$[x_1] = [x_0] \cap [y_0] - [f(x)_1]$	Backward-step
$y = f(x) + x$	$[y_1] = [y_0] \cap [f(x)_1] + [x_1]$	Backward-step

Experimental Evaluation

- SV-COMP 2021 benchmarks, standard PC desktop, TO = 900s

	ESBMC					
Benchmark	**w/o Contractor**	**with Contractor**	**search-space pruned**	**our approach (Contractor)**	**our approach time in sec**	**Frama-C**
afnp2014.c	Timeout	**2.03s**	99.99%	unknown	0.00187	successful
2dim.c	Timeout	**29.41s**	91%	unknown	4.4E-05	unknown
2dim-2.c	Timeout	**34.65s**	91%	unknown	2.3E-05	unknown
2dim-3.c	Timeout	**0.28s**	91%	unknown	2.2E-05	unknown
2dim-double-inter.c	Timeout	**80.24s**	93.20%	unknown	2.3E-05	unknown
bhmr2007.c	Timeout	**0.92s**	100%	successful	12.3338	unknown
ensure_order.c	848.01s	**174.26s**	100%	successful	0.00451	unknown
sum01-2.c	678.15s	**172.90s**	99.99%	unknown	2.0E-05	successful
arctan_Pade.c	212.82s	**0.35s**	100%	successful	0.0006	successful
gcd_2.c	58.48s	**48.19s**	0.79%	unknown	3.9E-05	unknown
gcd_3.c	76.09s	**46.42s**	0.79%	unknown	1.9E-05	unknown
verisec_...	0.50s	**0.27s**	100%	successful	1.4E-05	unknown

Induction-Based Verification for Software

```
k=1
while k<=max_iterations do
    if  base_{P,φ,k} then
        return trace s[0..k]
    else
      k=k+1
      if fwd_{P,φ,k} then
        return true
      else if step_{P',φ,k} then
        return true
    end if
end
return unknown
```

```
unsigned int x=*;
while(x>0) x--;
assume(x<=0);
assert(x==0);
```

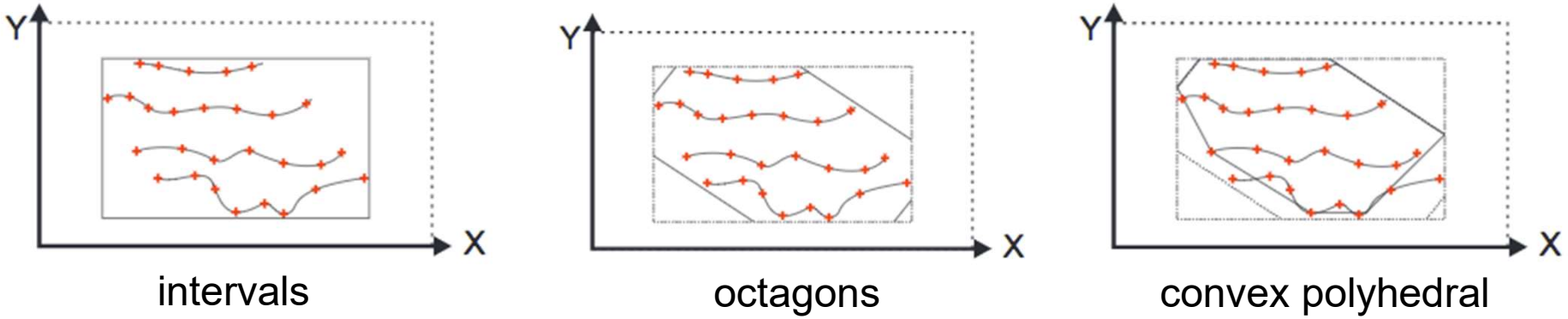
```
unsigned int x=*;
while(x>0) x--;
assert(x<=0);
assert(x==0);
```

```
unsigned int x=*;
assume(x>0);
while(x>0) x--;
assume(x<=0);
assert(x==0);
```

Gadelha et al.: Handling loops in bounded model checking of C programs via k-induction. STTT, 2017

Automatic Invariant Generation

- Infer invariants using **intervals**, **octagons**, and **convex polyhedral** constraints for the inductive step
  - *e.g.*, $a \le x \le b$; $x \le a$, $x\text{-}y \le b$; and $ax + by \le c$

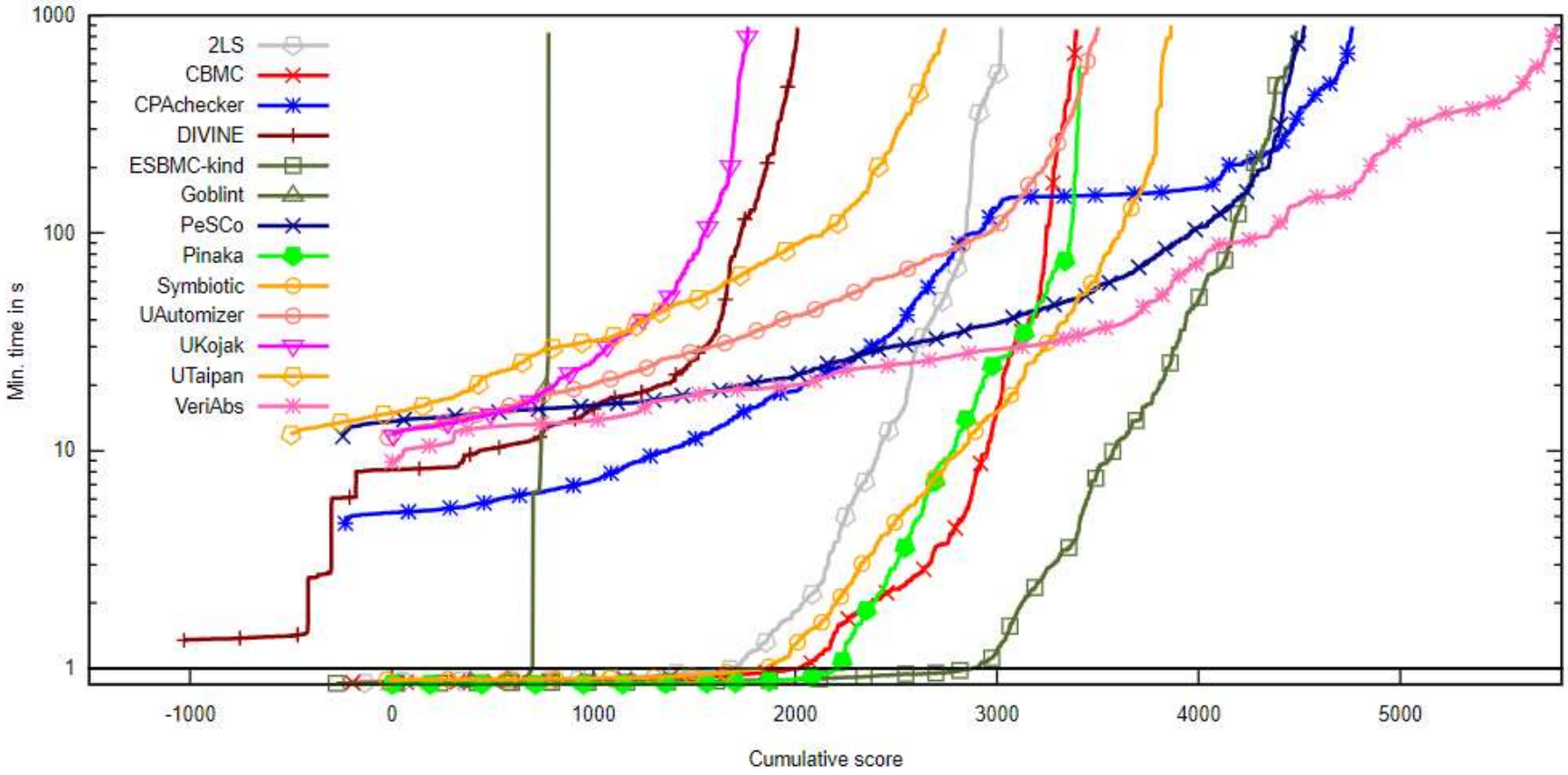


- Discover linear/polynomial relations among integer/real variables to infer **loop invariants**

_k_-Induction can prove the correctness of more programs when the invariant generation is enabled

Verification of the `Reach-Safety` Category

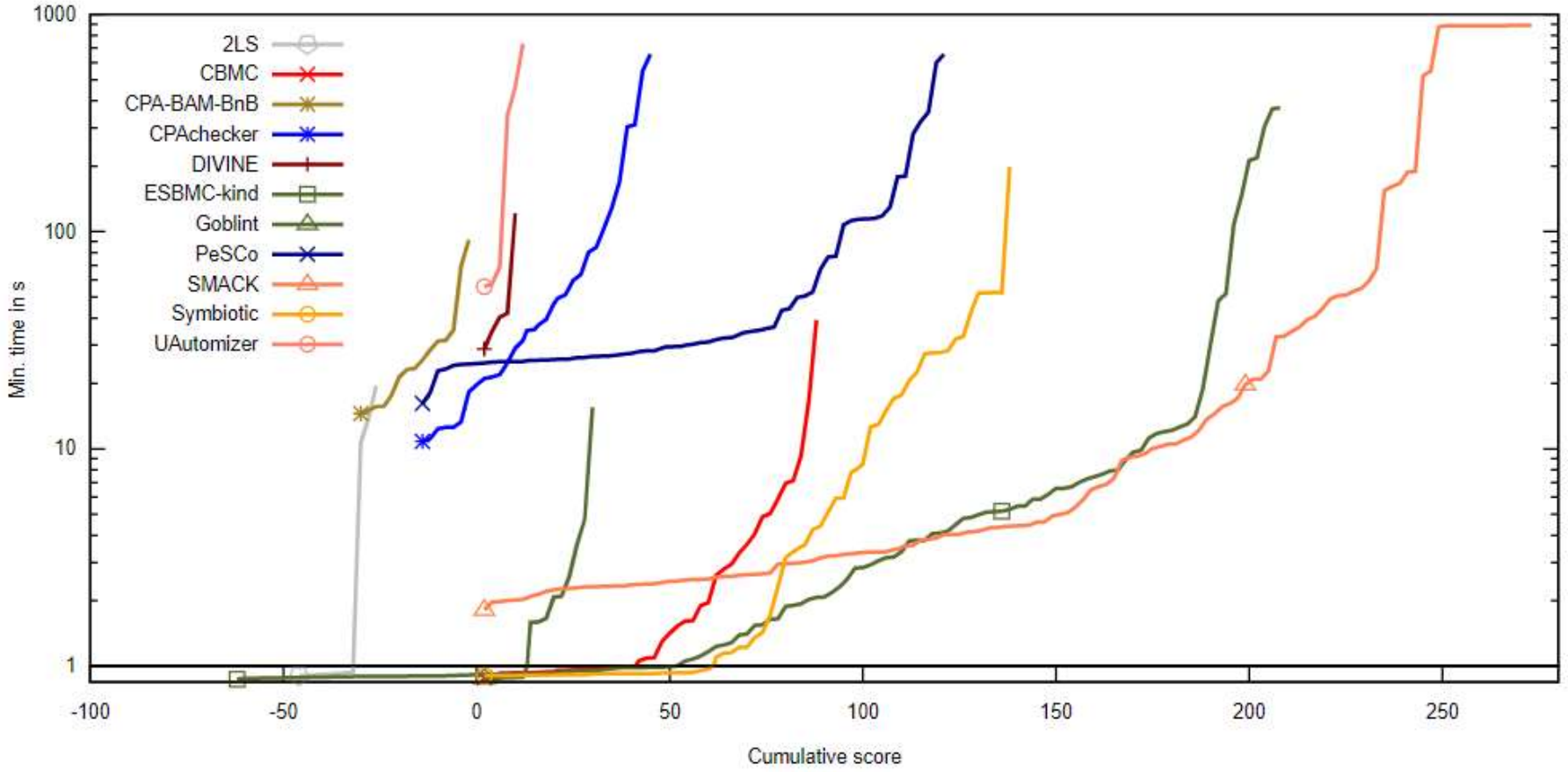
- SV-COMP 2021, 4927 verification tasks, max. score: 7844
- ESBMC achieved the 4th place



https://sv-comp.sosy-lab.org/2021/

Verification of the `AWS` Subcategory

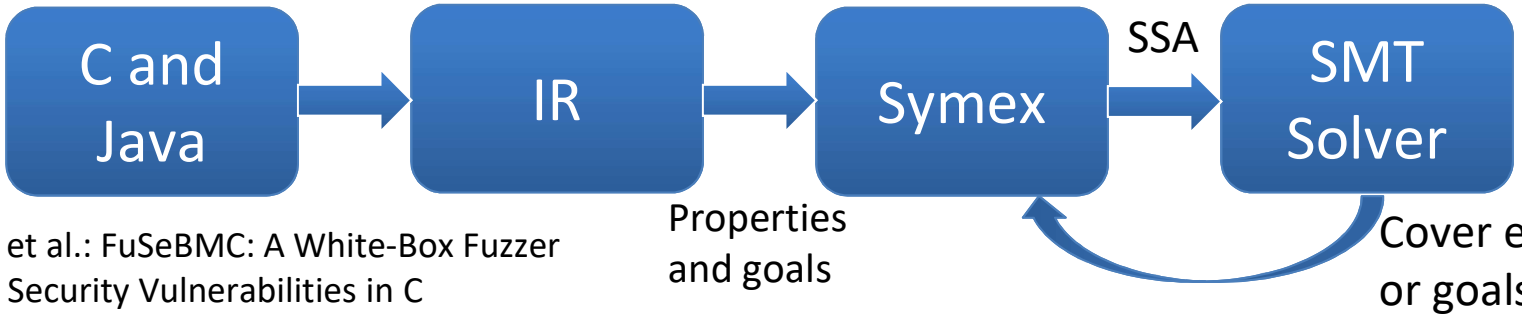
- SV-COMP 2021, 175 verification tasks, max. score: 345
- ESBMC achieved the 2th place



https://sv-comp.sosy-lab.org/2021/

BMC for Bug Finding and Code Coverage

- Translate the program to an **intermediate representation** (IR)
- Add safety properties to check for **errors** or goals to check for **coverage**
- **Symbolically** execute IR to produce an SSA program
- Translate the resulting SSA program into a **logical formula**
- Solve the formula iteratively to cover errors and goals
- Interpret the solution to figure out the **input conditions**
- Spit those input conditions out as a test case



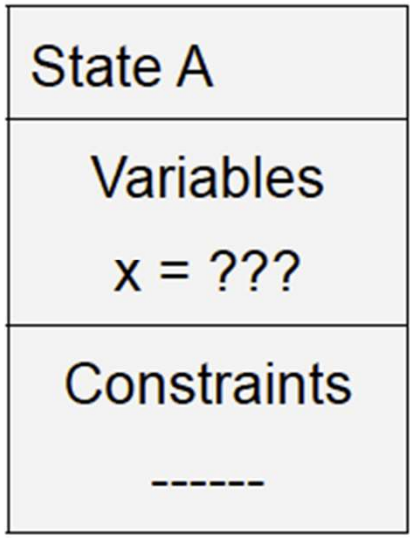
Alshmrany et al.: FuSeBMC: A White-Box Fuzzer for Finding Security Vulnerabilities in C Programs. FASE, 2021

Coverage Test Generation for Security

```
x = input();
if (x >= 10)
{
  if (x < 100)
    vulnerable_code();
  else
    func_a();
}
else
  func_b();
```

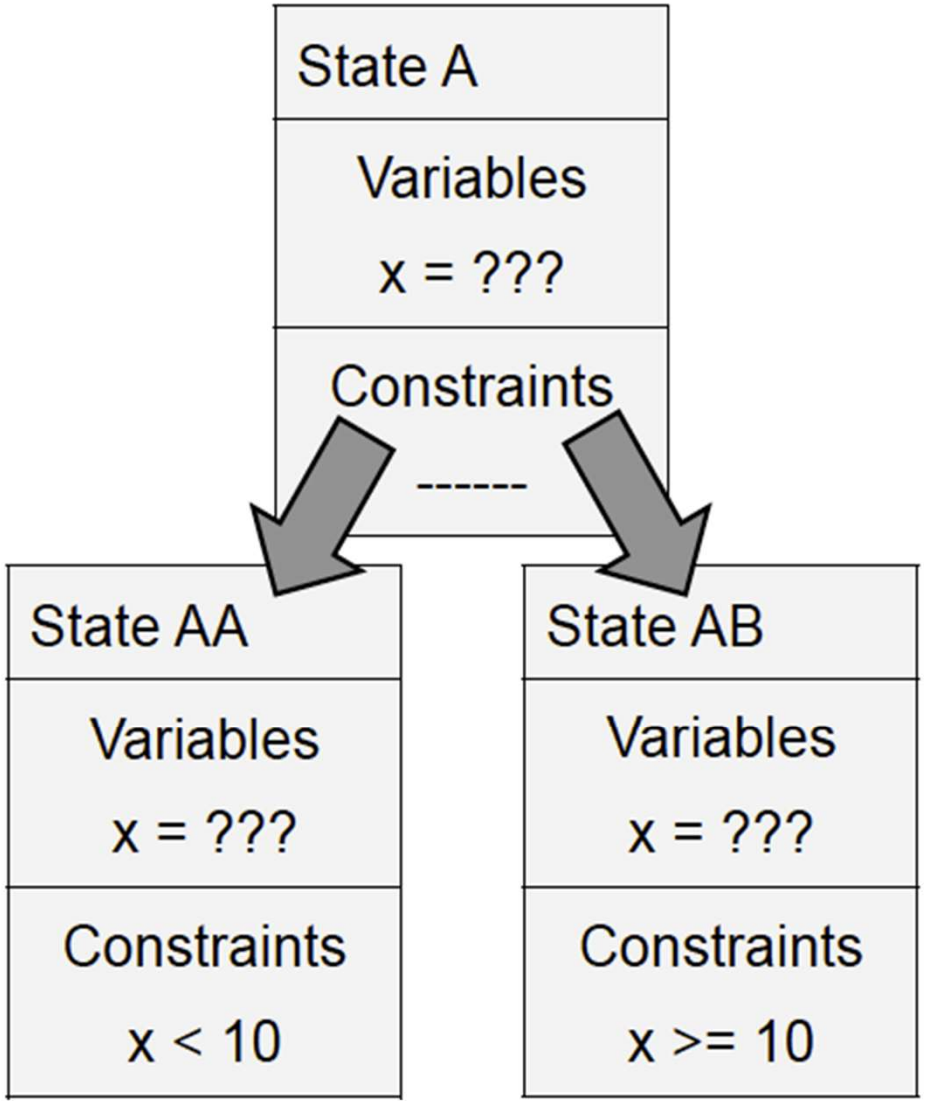
Coverage Test Generation for Security

```
x = input();
if (x >= 10)
{
  if (x < 100)
    vulnerable_code();
  else
    func_a();
}
else
  func_b();
```



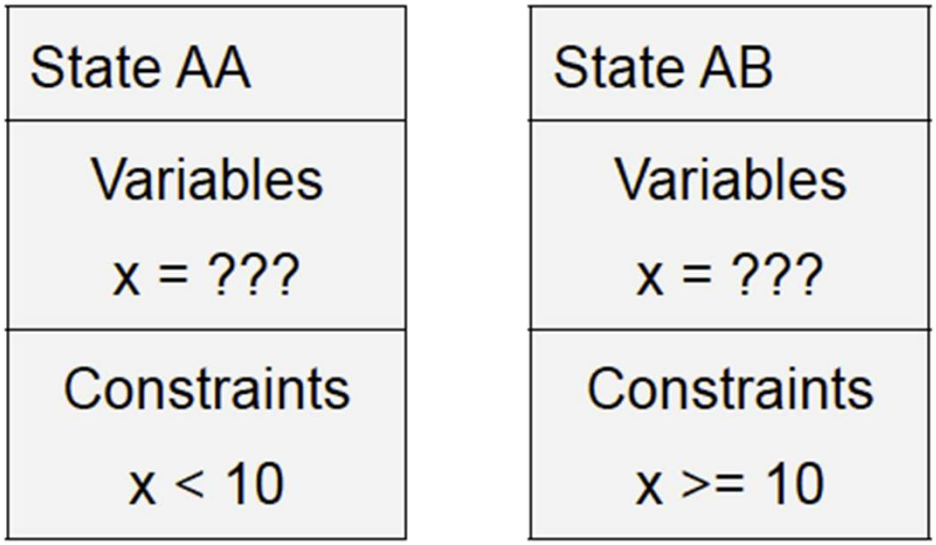
Coverage Test Generation for Security

```
x = input();
if (x >= 10)
{
  if (x < 100)
    vulnerable_code();
  else
    func_a();
}
else
  func_b();
```



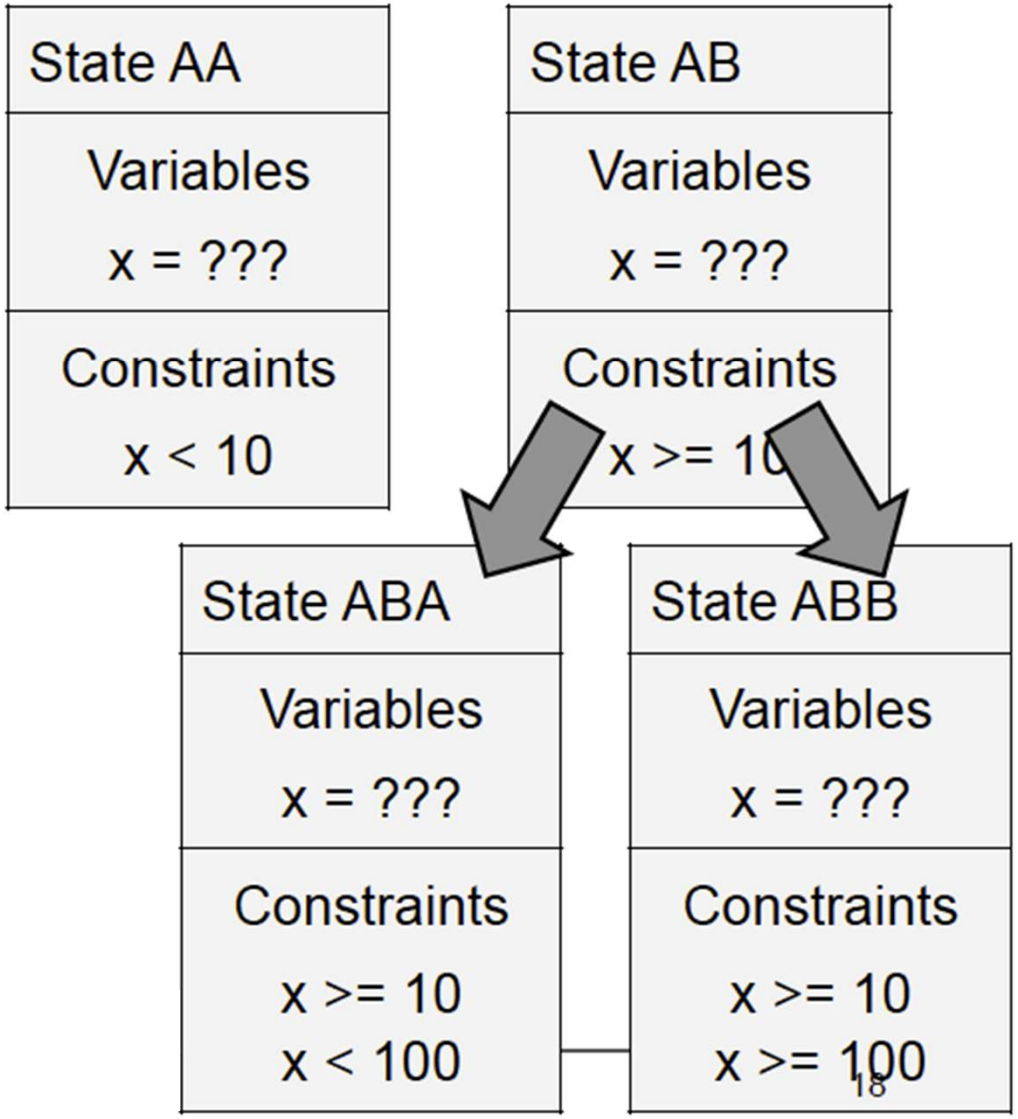
Coverage Test Generation for Security

```
x = input();
if (x >= 10)
{
  if (x < 100)
    vulnerable_code();
  else
    func_a();
}
else
  func_b();
```



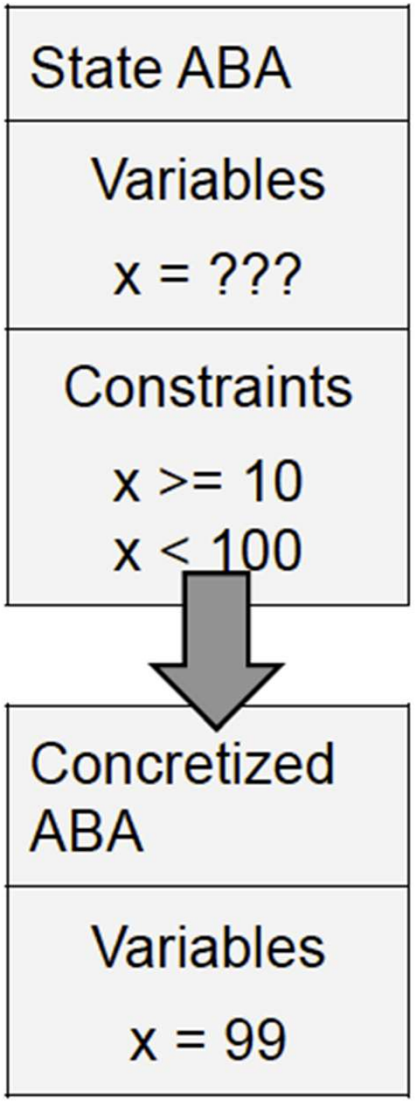
Coverage Test Generation for Security

```
x = input();
if (x >= 10)
{
  if (x < 100)
    vulnerable_code();
  else
    func_a();
}
else
  func_b();
```

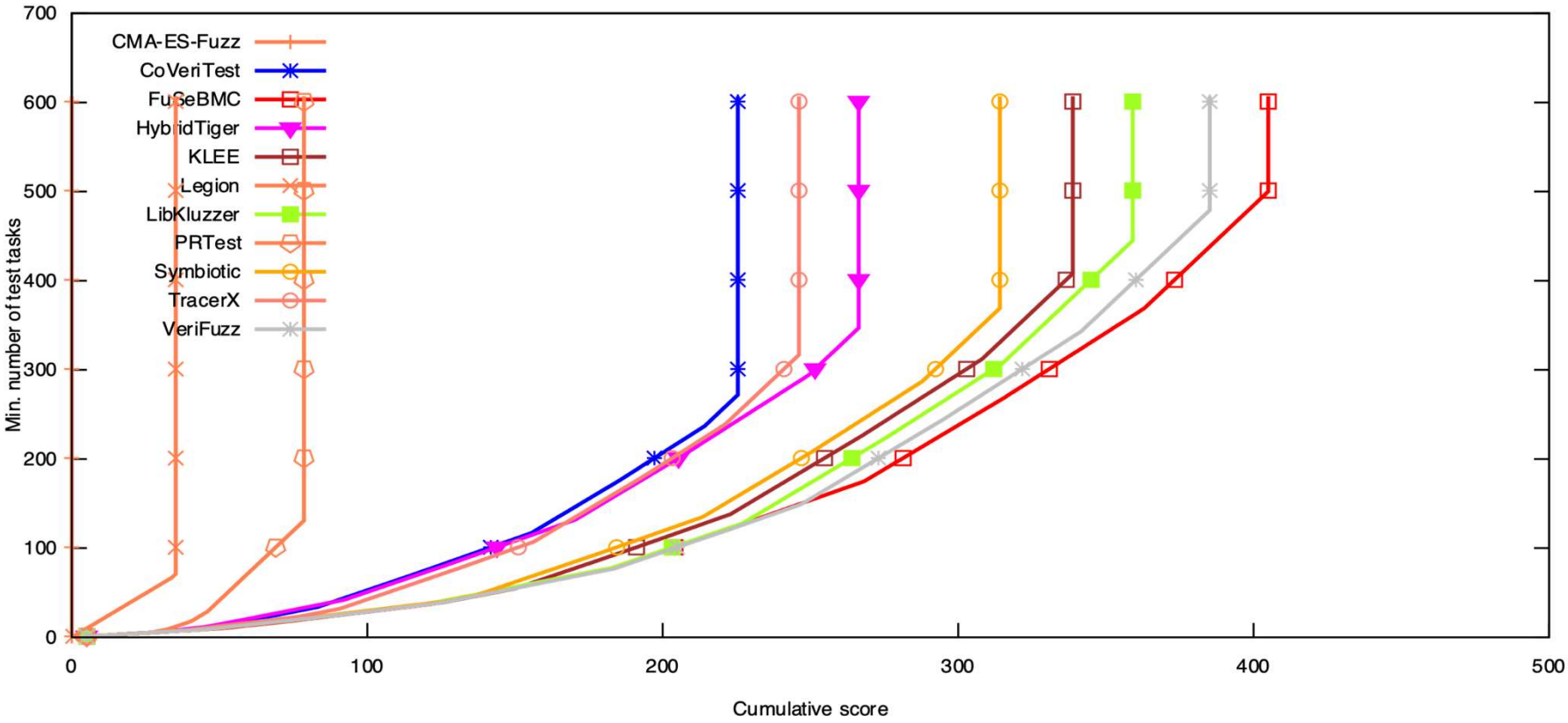


Coverage Test Generation for Security

```
x = input();
if (x >= 10)
{
  if (x < 100)
    vulnerable_code();
  else
    func_a();
}
else
  func_b();
```



Competition on Software Testing 2021: Results of the `Cover-Error` Category



FuSeBMC achieved 3 awards: 1st place in `Cover-Error`, 2nd place in `Overall`, and 3rd place in `Energy Consumption`

https://test-comp.sosy-lab.org/2021/

Achievements

- **Distinguished Paper Award** at ACM ICSE'11 (acceptance rate 14%)
- **28 awards** from the international competitions on software verification (SV-COMP) and testing (Test-Comp) 2012-2021 at **TACAS/FASE**
  - Bug finding
  - Cover error
- **Intel** deploys **ESBMC** in production as one of its verification engines for **verifying firmware in C**
- **Nokia** has found **security vulnerabilities** in **telecommunication software** written in **C++**

Conclusions

- **SAT/SMT-based software model checking** is a **competitive method** to verify programs
- We handle a variety of properties
  - Memory
  - Reachability
  - Concurrency
- We support incremental verification, k-induction, termination, and invariant inference
- The tools have been applied to find **security vulnerabilities** in **large-scale software systems**

Thank you

An Efficient SMT-based Bounded Model Checker

GitHub

Documentation
News
Publications
SV-COMP
Test-Comp
People
Applications
Download Archive
Third Party Contributions
Index of Benchmarks

ESBMC is an open source, permissively licensed, context-bounded model checker based on satisfiability modulo theories for the verification of single- and multi-threaded C/C++ programs. It does not require the user to annotate the programs with pre- or postconditions, but allows the user to state additional properties using assert-statements, that are then checked as well. Furthermore, ESBMC provides two approaches (lazy and schedule recording) to model check multi-threaded programs. It converts the verification conditions using different background theories and passes them directly to an SMT solver.

ESBMC is a joint project with the Federal University of Amazonas, University of Bristol, University of Manchester, University of Stellenbosch, and University of Southampton.

News

21/03/2021: ESBMC v6.7 for Linux, macOS and Windows released.

30/12/2020: ESBMC v6.6 for Linux and MacOS released.

30/12/2020: ESBMC has successfully participated at the 10th Intl. Competition on Software Verification held at TACAS 2021 in Luxembourg. ESBMC won first place in the **ReachSafety-XCSP** subcategory. Second place in the **SoftwareSystems-AWS-C-Common-ReachSafety**, **ReachSafety-ECA**, and

http://esbmc.org/
https://github.com/esbmc/esbmc