Collaborators/funders:

- Systems and Software Security / FM Research Group - UoM
- ARM Centre of Excellence - UoM
- PPGEE, PPGI – UFAM
- Centre for Digital Trust and Society - UoM
- UKRI, EPSRC, EU Horizon, and industrial partners

SBMF 2023
26th Brazilian Symposium on Formal Methods

Exploring Automated Software Testing, Verification, and Repair Strategies

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

Career Summary

1,7

BSc/MSc in Engineering and Lecturer

2

MSc in Embedded Systems

3

SIEMENS mobile

Configuration and Build Manager

4

BenQ SIEMENS

Feature Leader

5

NXP

Set-top Box Software Engineer

6

PhD in Computer Science

8

Postdoctoral Researcher

9

MANCHESTER 1824
The University of Manchester

Reader in Program Analysis and Cyber-Security

Systems and Software Security Research Group

L. Cordeiro

R. Banach

M. Mustafa

N. Zhang

B. Magri

Y. Sun

D. Dresner

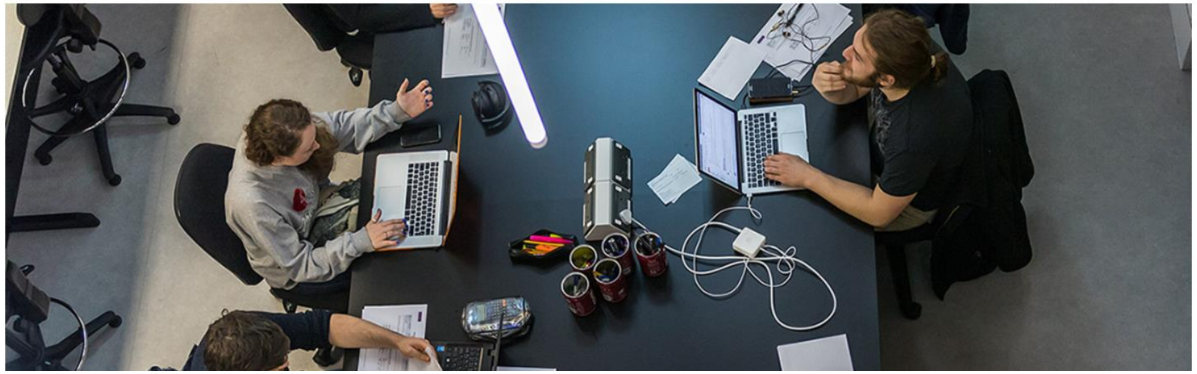
A. Creswell

ARM Centre of Excellence

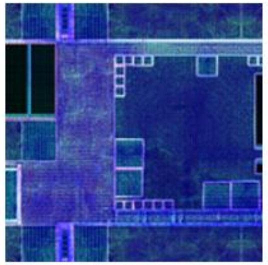
Department of Computer Science / Arm Centre of Excellence

Arm Centre of Excellence

Our Centre of Excellence (CoE) expertise covers the full spectrum of modern processor technologies, machine learning theory and applications, and automation of reasoning to formally build verifiable trustworthy hardware and software systems.



https://www.cs.manchester.ac.uk/arm-coe/



Advanced processor technologies

We investigate, develop and evaluate advanced and novel approaches to processing and computation, emphasising identifying novel ways to exploit the formidable complexity of the multi-billion transistor microchips available today.

Machine learning and robotics

We investigate and develop modern dimensionality reduction methods, preserving local and global structure relationships. We also scientifically contribute to the foundations of information-theoretic variable selection and state of the art Deep Learning methodologies applied in image and video understanding.

Systems and software security

We develop algorithms, methods and protocols to address security and privacy in distributed embedded applications and tools to build verifiable, trustworthy software systems. Our expertise covers a broad span of topics, including digital trust, security, and privacy.

Centre for Digital Trust and Society

https://www.socialsciences.manchester.ac.uk/dts/

Clusters

Digital Technologies and Crime

Focuses on analysing and understanding criminal activity across the digital spectrum.

Trusted Digital Systems

Developing tools, policies and practices to safeguard data and communication.

Workplace and Organisational Security

Researching institutional security, insider threats, supply chain security and psychology of crime.

Democracy and Trust

Researching the threats that digital technologies and AI pose to trust and security in democratic systems.

Privacy and Trust

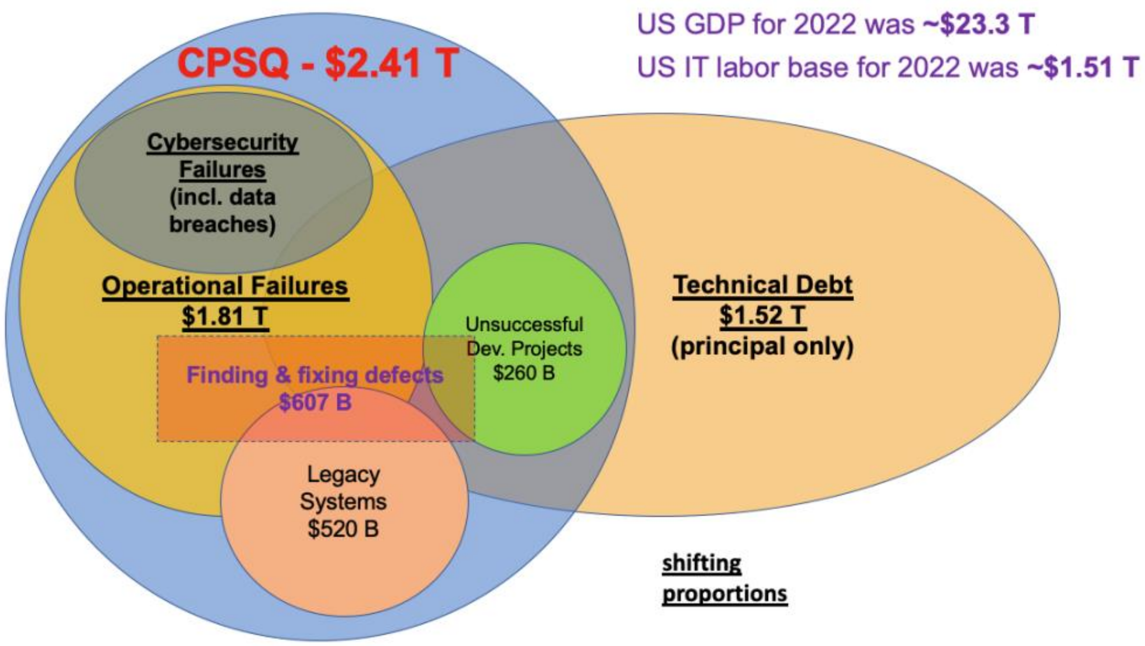
Focusing on the interplay of a complex number of topics around privacy and confidentiality.

Advanced Mathematics

Applying advanced mathematical theory and methods for cyber security.

How much could software errors cost your business?

Poor software quality cost US companies **$2.41 trillion in 2022**, while the **accumulated software Technical Debt** (TD) has grown to **~$1.52 trillion**



TD relies on temporary easy-to-implement solutions to achieve short-term results at the expense of efficiency in the long run

The cost of poor software quality in the US: A 2022 Report

CISQ Consortium for Information & Software Quality™

Objective of this talk

Discuss **automated testing, verification, and repair** techniques to establish a **robust foundation for building secure software systems**

- Introduce a **logic-based automated reasoning platform** to find and repair **software vulnerabilities**
- Explain **testing, verification, and repair** techniques to build **secure software systems**
- Present recent advancements towards a **hybrid approach** to protecting against **memory safety and concurrency vulnerabilities**

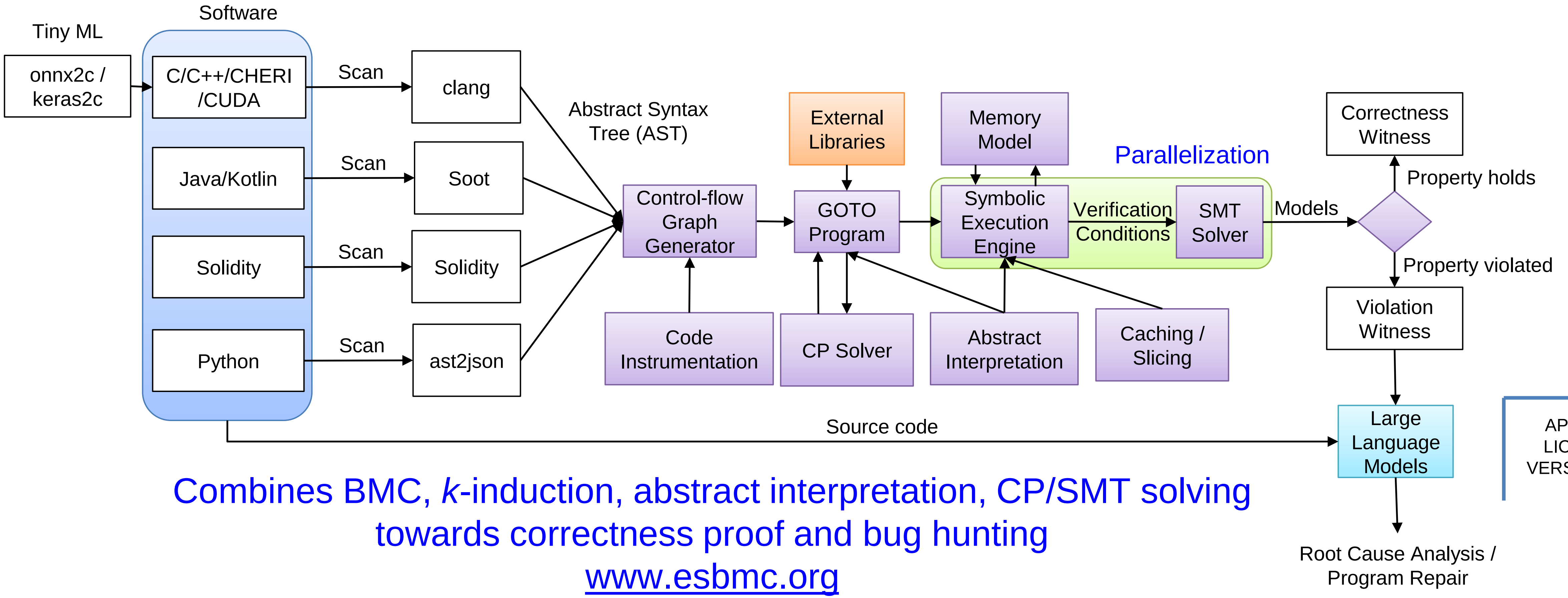
Research Questions

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

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

ESBMC: An Automated Verification Platform

Logic-based automated reasoning for checking the **safety** and **security** of **AI and software systems**



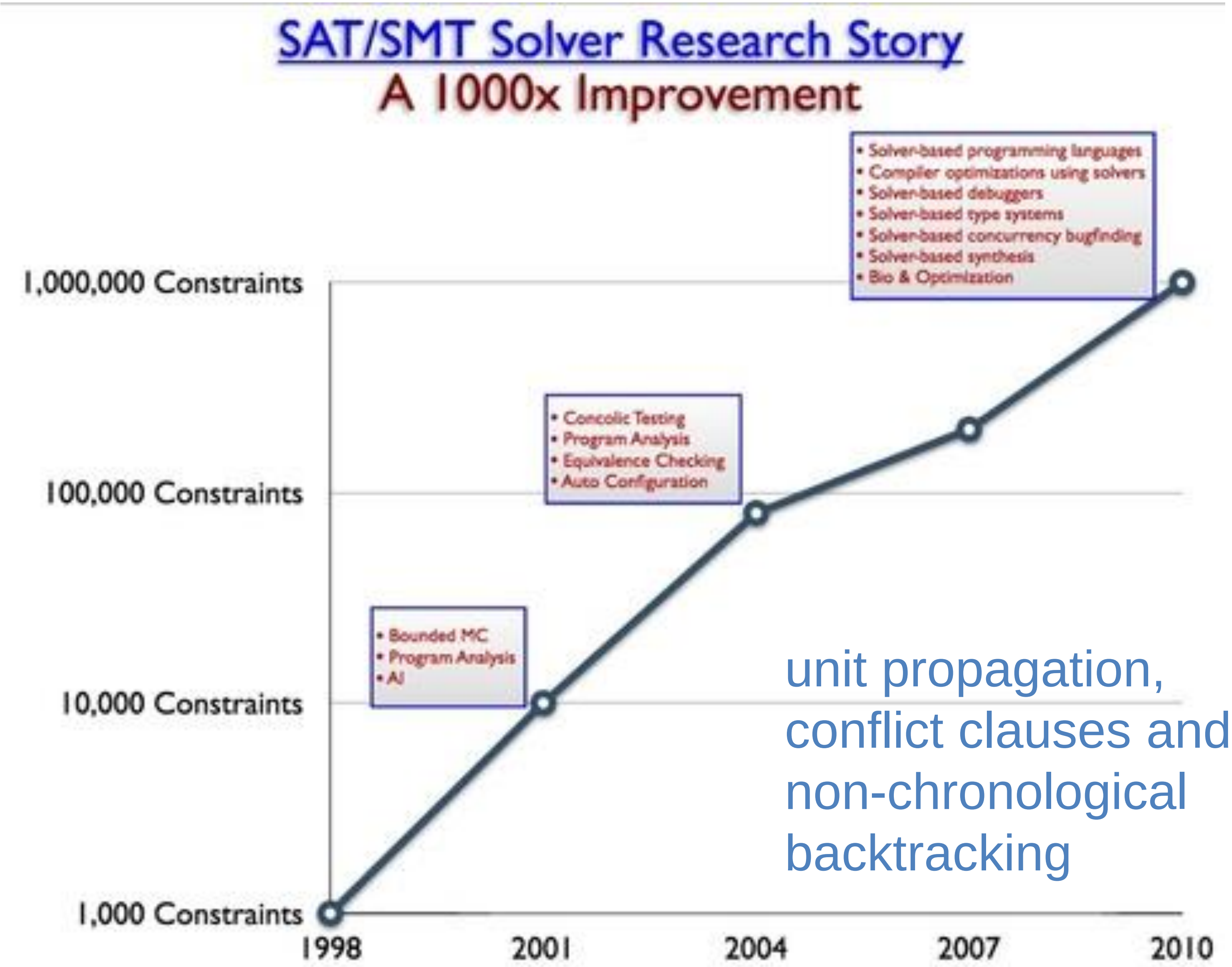
Combines BMC, *k*-induction, abstract interpretation, CP/SMT solving towards correctness proof and bug hunting

www.esbmc.org

Agenda

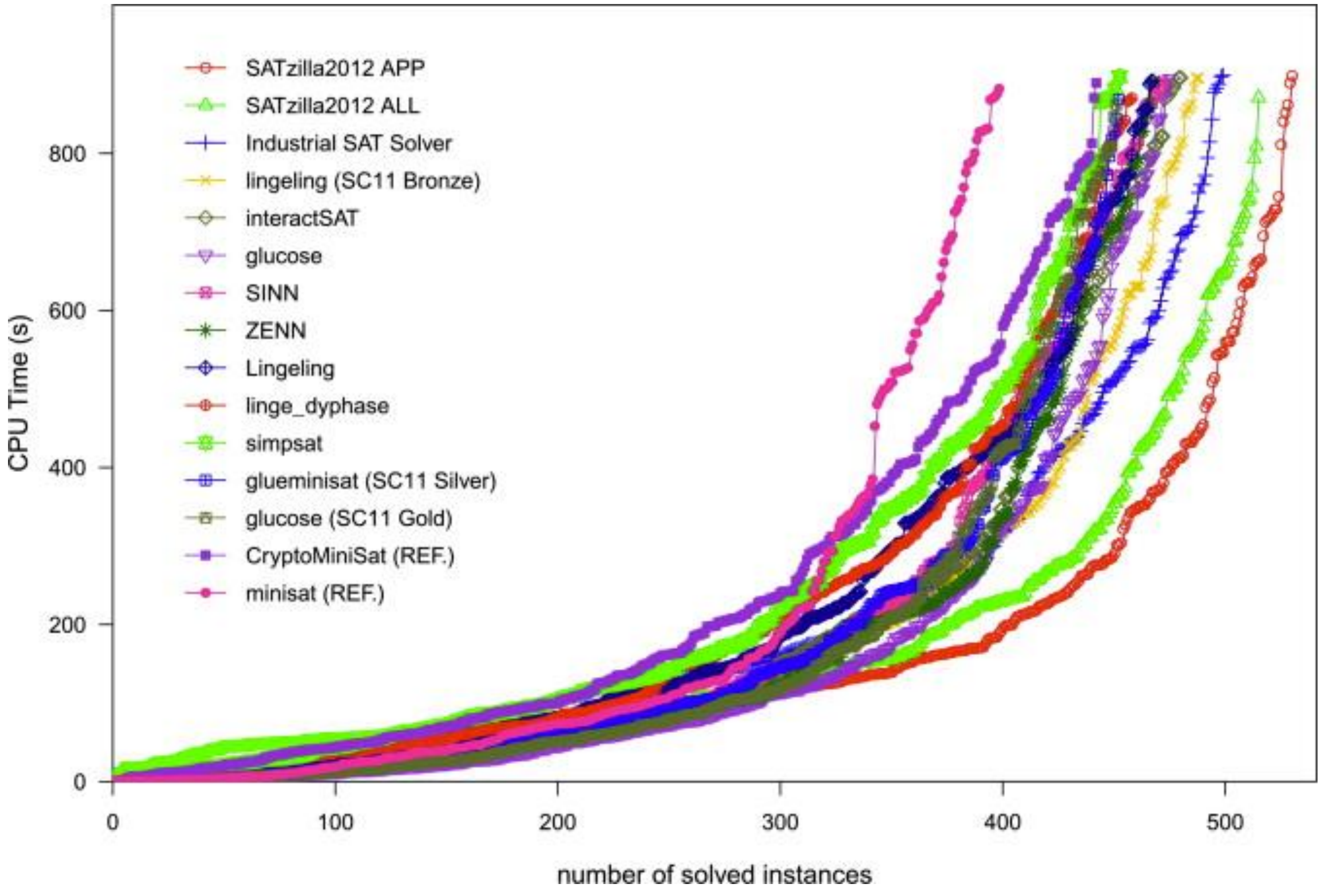
- Intoduce typical BMC Architectures for Verifying Software Systems
- Software Verification and Testing with the ESBMC Framework
- Towards Self-Healing Software via Large Language Models and Formal Verification
- Towards Verification of Programs for CHERI Platforms with ESBMC

SAT solving as enabling technology



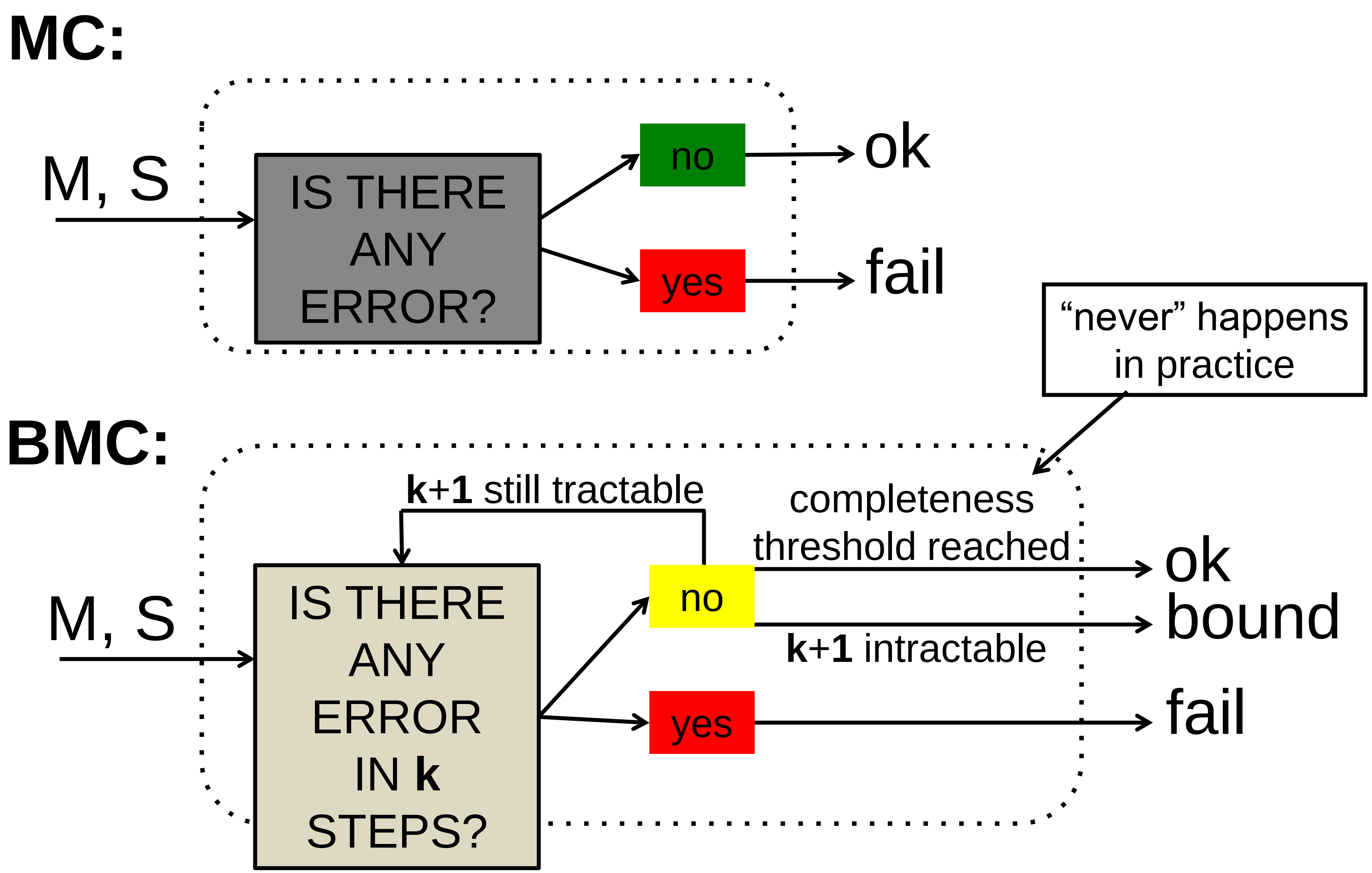
unit propagation, conflict clauses and non-chronological backtracking

SAT Competition



http://www.satcompetition.org/

Bounded Model Checking (BMC)



SAT/SMT-based BMC tools for C

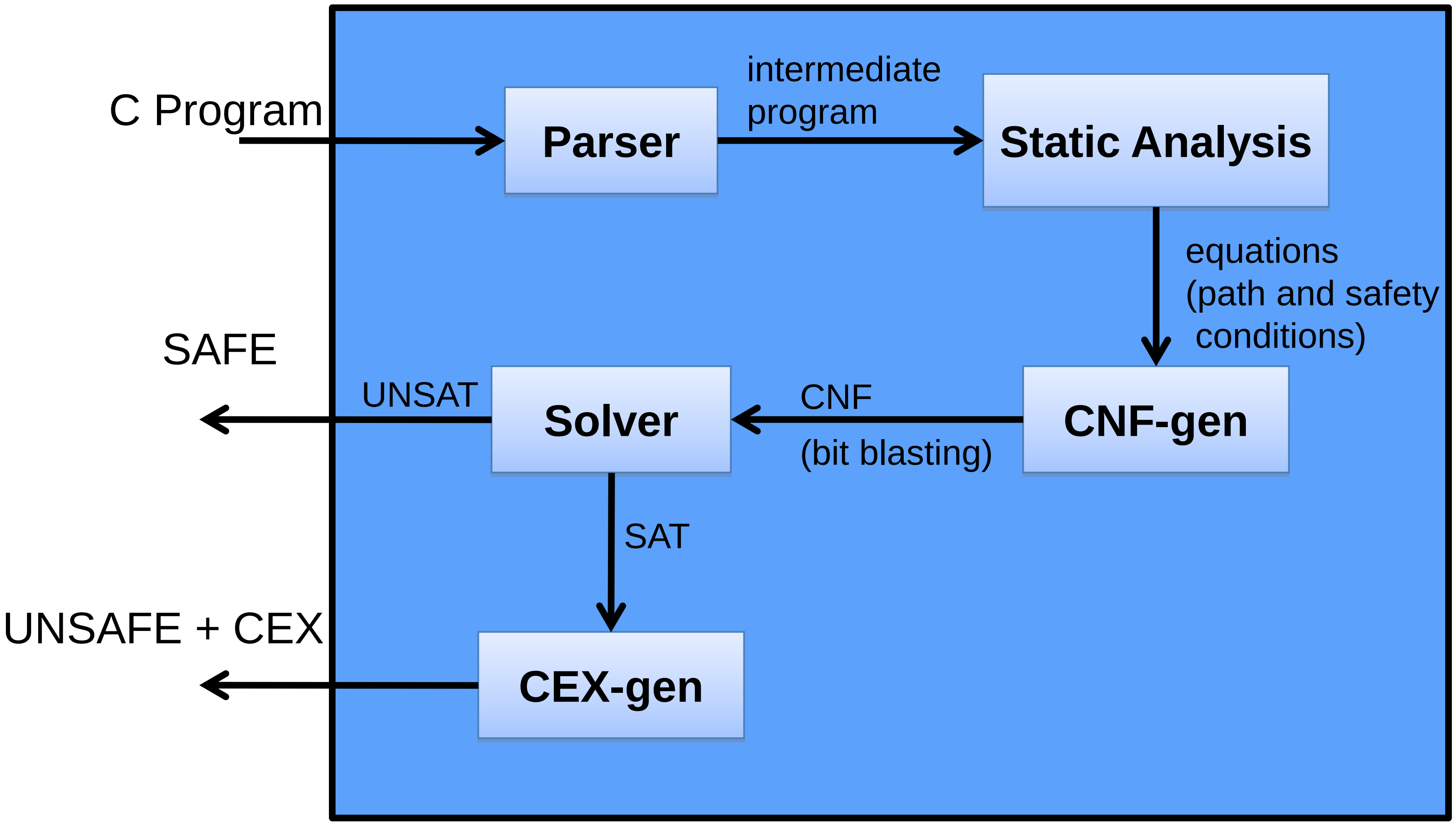
- **CBMC (C Bounded Model Checker)**
  - http://www.cprover.org/
  - SAT-based (MiniSat) “workhorse”
  - also SystemC frontend
- **ESBMC (The Efficient SMT-based Bounded Model Checker)**
  - http://esbmc.org
  - SMT-based (Z3, Boolector, Yices, Bitbuwzla, MathSAT, etc)
  - Clang frontend, Soot, Solidity, and Python
- **LLBMC (Low-level Bounded Model Checker)**
  - http://llbmc.org
  - SMT-based (Boolector or STP)
  - uses LLVM intermediate language

⇒share common high-level architecture

Typical Features for BMC Architectures

- **Full language support**
  - bit-precise operations, structs, arrays, ...
  - heap-allocated memory
  - concurrency
- **Built-in safety checks**
  - overflow, div-by-zero, array out-of-bounds indexing, ...
  - memory safety: nil pointer deref, memory leaks, ...
  - deadlocks, race conditions
- **User-specified assertions and error labels**
- **Non-deterministic modelling**
  - nondeterministic assignments
  - assume-statements

High-Level BMC Architectures



General Approach

1. Simplify control flow
2. Unwind all of the loops
3. Convert into single static assignment (SSA) form
4. Convert into equations and simplify
5. (Bit-blast)
6. Solve with a SAT/SMT solver
7. Convert SAT assignment into a counterexample

Control flow simplifications

- remove all side effects
  - e.g., j=++i; becomes i=i+1; j=i;
- simplify all control flow structures into core forms
  - e.g., replace for, do while by while
  - e.g., replace case by if
- make control flow explicit
  - e.g., replace continue, break by goto
  - e.g., replace if, while by goto

Control flow simplifications

Demo: esbmc --goto-functions-only example-1.c

```
int main() {
  int i,j;
  for(i=0; i<6; i++) {
   j=i;
  }
  assert(j==i);
  return j;
}
```

```
int main() {
  int i,j;
  i=0;
  while(i<6) {
   j=i;
   i++;
  }
  assert(j==i);
  return j;
}
```

```
main (c::main):
        int i;
        int j;
  i++   i = 0;
    1: IF !(i < 6) THEN GOTO 2
        j = i;
        i = i + 1;
      GOTO 1
    2:   ASSERT j == i
      RETURN: j
      END_FUNCTION
```

Control flow simplifications

```
main (c::main):
        int i;
        int j;
        i = 0;
    1: IF !(i < 6) THEN GOTO 2
        j = i;
        i = i + 1;
      GOTO 1
    2:   ASSERT j == i
      RETURN: j
      END_FUNCTION
```

$C := i_1 = 0 \wedge$
$g1 = (i_1 >= 6)\ ?\ true : false \wedge$
$j_1 = g_1\ ?\ j_0 : i_1 \wedge$
$i_2 = g_1\ ?\ i_1 + 1 \wedge$
$g_2 = (i_2 >= 6)\ ?\ true : false \wedge$
$j_2 = g_2\ ?\ j_1 : i_2 \wedge$
$i_2 = g_2\ ?\ i_1 + 1 \wedge$
…
$g_6 = (i_6 >= 6)\ ?\ true : false \wedge$
$j_6 = g_6\ ?\ j_5 : i_6 \wedge$
$i_6 = g_6\ ?\ i_5 + 1 \wedge$
$return_1 = j_6$

$P := (j_6 == i_6)$

Loop unwinding

- All loops are “unwound”, i.e., replaced by several guarded copies of the loop body
  - same for backward `gotos` and recursive functions
  - can use different unwinding bounds for different loops

⇒ Each statement is executed at most once

- to check whether unwinding is sufficient special “unwinding assertion” claims are added

⇒ If a program satisfies all of its claims and all unwinding assertions then it is correct!

Loop unwinding

```
void f(...) {
  ...
  while(cond) {
    Body;
  }
  Remainder;
}
```

Loop unwinding

```
void f(...) {
  ...
  if(cond) {
    Body;
    while(cond) {
      Body;
    }
  }
  Remainder;
}
```

unwind one iteration

Loop unwinding

```
void f(...) {
  ...
  if(cond) {
    Body;
    if(cond) {
      Body;
      while(cond) {
        Body;
      }
    }
  }
  Remainder;
}
```

unwind one

unwind one
iteration

Loop unwinding

```
void f(...) {
  ...
  if(cond) {
    Body;
    if(cond) {
      Body;
      if(cond) {
        Body;
        while(cond) {
          Body;
        }
      }
    }
  }
  Remainder;
}
```

unwind one

unwind one

unwind one iteration…

Loop unwinding

```
void f(...) {
  ...
  if(cond) {
    Body;
    if(cond) {
      Body;
      if(cond) {
        Body;
        assert(!cond);

      }
    }
  }
  Remainder;
}
```

unwind one iteration…

unwinding assertion

- unwinding assertion
  - inserted after last unwound iteration
  - violated if program runs longer than bound permits

  ⇒ if not violated: (real) correctness result!

Loop unwinding

```
void f(...) {
  ...
  for(i=0; i<N; i++) {
    ...
    b[i]=a[i];
    ...
  };
  ...
  for(i=0; i<N; i++) {
    ...
    assert(b[i]-a[i]>0);
    ...
  };
  ...
  Remainder;
}
```

- unwinding assertion
  - inserted after last unwound iteration
  - violated if program runs longer than bound permits
  - ⇒ if not violated: (real) correctness result!
- ⇒ what about multiple loops?
  - use --partial-loops to suppress insertion
  - ⇒ unsound

Safety conditions

- Built-in safety checks converted into explicit assertions:

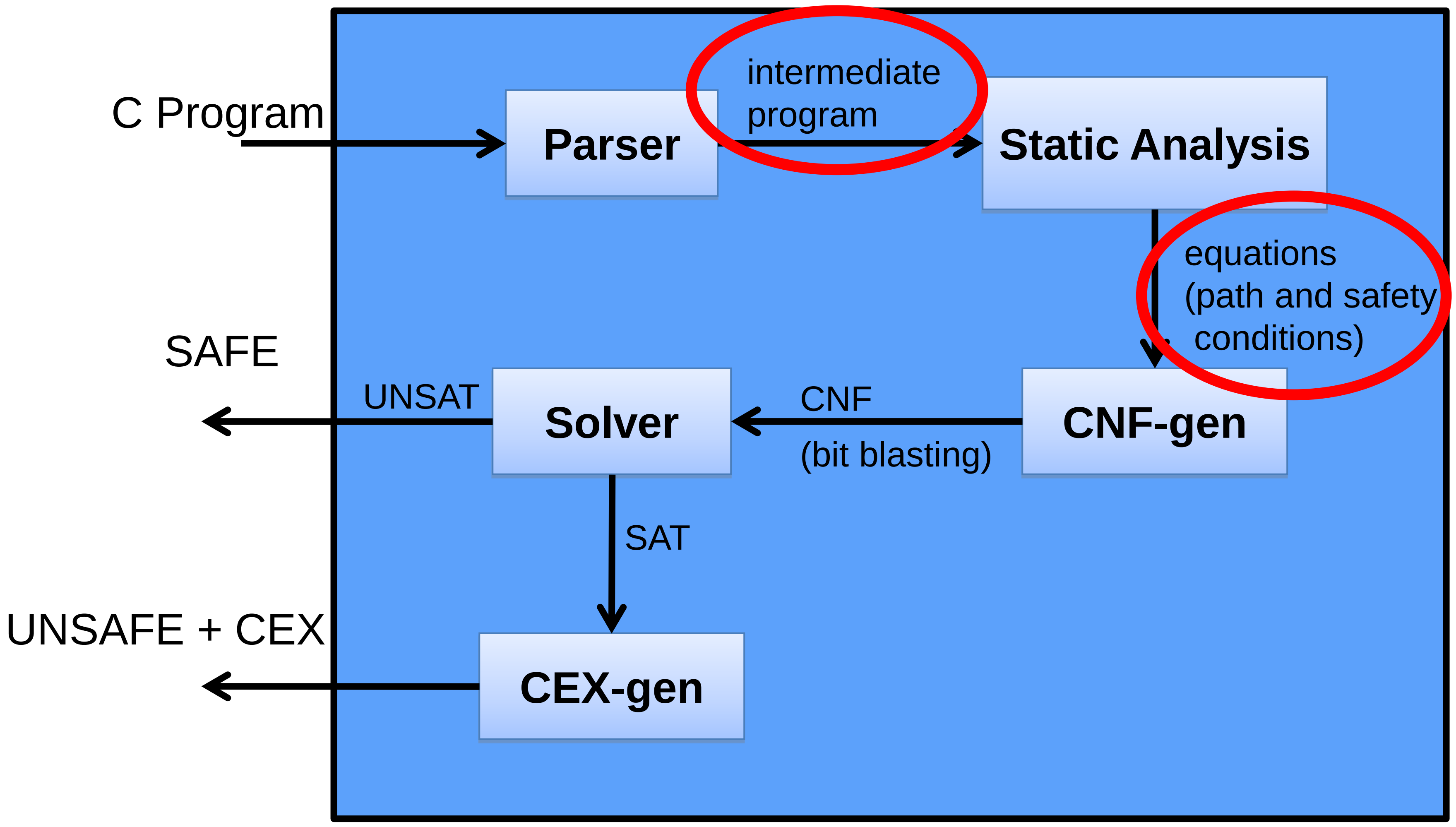
  e.g., array safety:

  a[i]=...;
  ⇒ assert(0 <= i && i < N); a[i]=...;

⇒ sometimes easier at intermediate representation or formula level

  e.g., word-aligned pointer access, overflow, ...

High-Level Architecture



Transforming straight-line programs into equations

- simple if each variable is assigned only once:

```
x = a;
y = x + 1;
z = y – 1;
```

program

→

```
x = a          &&
y = x + 1      &&
z = y – 1
```

constraints

- still simple if variables are assigned multiple times:

```
x = a;
x = x + 1;
x = x – 1;
```

program

→

$x_0 = a;$
$x_1 = x_0 + 1;$
$x_2 = x_1 - 1;$

program in SSA-form

introduce fresh copy for each occurrence (*static single assignment (SSA)* form)

Transforming loop-free programs into equations

But what about control flow branches (if-statements)?

```
if(v)
  x = y;
else
  x = z;

w = x;
```

```
if(v_0)
  x_0 = y_0;
else
  x_1 = z_0;

w_1 = ?
```

introduce & use new variable

- for each control flow join point, add a new variable with guarded assignment as definition
  - also called ϕ-function

Transforming loop-free programs into equations

But what about control flow branches (if-statements)?

```
if(v)
  x = y;
else
  x = z;

w = x;
```

$\texttt{if}(v_0)$
$\quad x_0 = y_0;$
$\texttt{else}$
$\quad x_1 = z_0;$
$x_2 = v_0 \ ?\ x_0 : x_1;$
$w_1 = x_2;$

introduce & use new variable

- for each control flow join point, add a new variable with guarded assignment as definition
  - also called ϕ-function

Bit-blasting

Conversion of equations into SAT problem:

- simple assignments:

  $|[\ x = y\ ]| \triangleq \bigwedge_i x_i \Leftrightarrow y_i$

  effective bitwidth

  $\Rightarrow$ static analysis must approximate effective bitwidth well

- ϕ-functions:

  $|[\ x = v\ ?\ y : z\ ]| \triangleq (v \Rightarrow |[\ x = y\ ]|) \wedge (\neg\ v \Rightarrow |[\ x = z\ ]|)$

- Boolean operations:

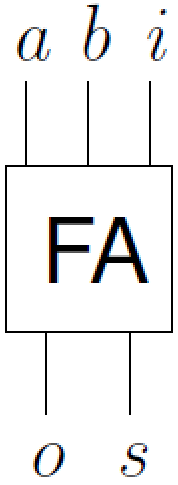
  $|[\ x = y \mid z\ ]| \triangleq \bigwedge_i x_i \Leftrightarrow (y_i \vee z_i)$

Exercise: relational operations

Bit-blasting arithmetic operations

Build **circuits** that implement the operations!

1-bit addition:



Full Adder

$$s \equiv (a + b + i) \bmod 2 \equiv a \oplus b \oplus i$$

$$o \equiv (a + b + i) \operatorname{div} 2 \equiv a \cdot b + a \cdot i + b \cdot i$$

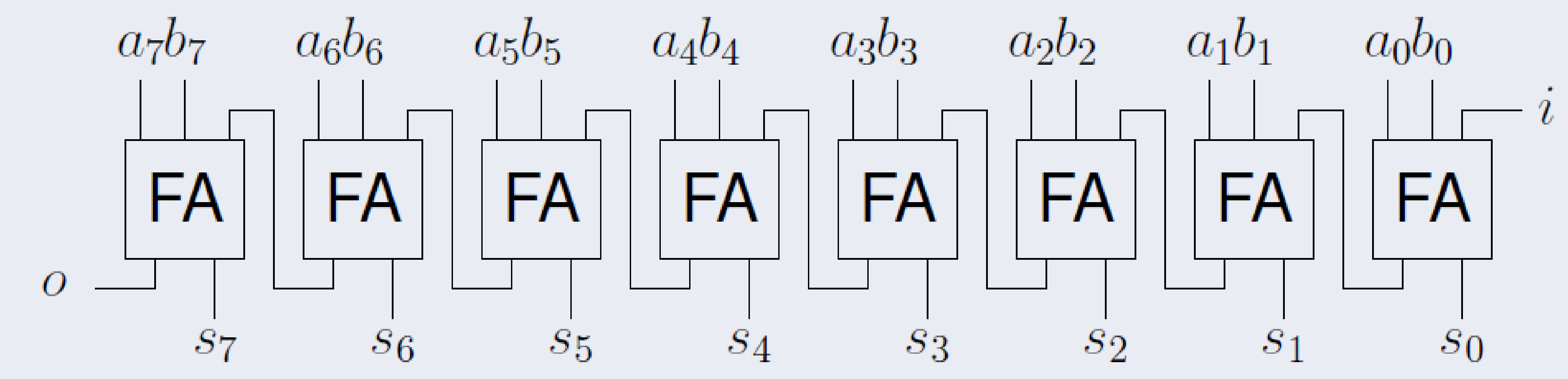
Full adder as CNF:

$$(a \vee b \vee \neg o) \wedge (a \vee \neg b \vee i \vee \neg o) \wedge (a \vee \neg b \vee \neg i \vee o) \wedge$$
$$(\neg a \vee b \vee i \vee \neg o) \wedge (\neg a \vee b \vee \neg i \vee o) \wedge (\neg a \vee \neg b \vee o)$$

Bit-blasting arithmetic operations

Build **circuits** that implement the operations!

8-Bit ripple carry adder (RCA)



⇒adds w variables, 6*w clauses

⇒multiplication / division much more complicated

Handling Arrays

Arrays can be replaced by individual variables, with a “demux” at each access:

```
int a[10];
...
x = a[i];
```

```
int a0, a1, a2, ... a9;
...
x = (i==0 ? a0
      : (i==1 ? a1
         : (i==2 ? a2
            : ...);
```

⇒surprisingly effective (for N<1000) because value of *i* can often be determined statically

– due to constant propagation

Handling Arrays with Theories

Arrays can be seen as ADT with two operations:

- read: Array x Index → Elem *"select"*
- write: Array x Index x Element → *"update"*

```
...
a[i]=a[i]+1;
...
```

⇒

```
...
a₁=write(a₀,i,read(a₀,i)+1);
...
```

Axioms describe intended semantics:

a write modifies the position written to ...

$$p = r \implies \mathtt{read}(\mathtt{write}(a, p, v), r) = v$$

$$\neg(p = r) \implies \mathtt{read}(\mathtt{write}(a, p, v), r) = \mathtt{read}(a, r)$$

...and nothing else

⇒requires support by **SMT-solver**

SAT vs. SMT

BMC tools use both propositional satisfiability (SAT) and satisfiability modulo theories (SMT) solvers:

- SAT solvers require encoding everything in CNF
  - limited support for high-level operations
  - easier to reflect machine-level semantics
  - can be extremely efficient (SMT falls back to SAT)
- SMT solvers support built-in theories
  - equality, free function symbols, arithmetics, arrays,...
  - sometimes even quantifiers
  - very flexible, extensible, front-end easier
  - requires extra effort to enforce precise semantics
  - can be slower

Satisfiability Modulo Theories

- SMT decides the **satisfiability** of first-order logic formulae using the combination of different **background theories** (building-in operators)

Theory	**Example**
Equality	$x_1=x_2 \land \neg (x_1=x_3) \Rightarrow \neg(x_1=x_3)$
Bit-vectors	(b >> i) & 1 = 1
Linear arithmetic	$(4y_1 + 3y_2 \geq 4) \lor (y_2 - 3y_3 \leq 3)$
Arrays	$(j = k \land a[k]=2) \Rightarrow a[j]=2$
Combined theories	$(j \leq k \land a[j]=2) \Rightarrow a[i] < 3$

Satisfiability Modulo Theories

- let **a** be an array, **b**, **c** and **d** be signed bit-vectors of width 16, 32 and 32 respectively, and let **g** be an unary function.

$$g(select(store(a,c,12)), SignExt(b,16)+3) \neq g(SignExt(b,16)-c+4) \wedge SignExt(b,16) = c-3 \wedge c+1 = d-4$$

⇩ **b'** extends **b** to the signed equivalent bit-vector of size 32

$$step1: g(select(store(a,c,12), b'+3)) \neq g(b'-c+4) \wedge b' = c-3 \wedge c+1 = d-4$$

⇩ replace b' by c−3 in the inequality

$$step2: g(select(store(a,c,12), c-3+3)) \neq g(c-3-c+4) \wedge c-3 = c-3 \wedge c+1 = d-4$$

⇩ using facts about bit-vector arithmetic

$$step3: g(select(store(a,c,12), c)) \neq g(1) \wedge c-3 = c-3 \wedge c+1 = d-4$$

Satisfiability Modulo Theories

$$step\ 3: g(select(store(a, c, 12), c)) \neq g(1) \wedge c - 3 = c - 3 \wedge c + 1 = d - 4$$

⇩ applying the theory of arrays

$$step\ 4: g(12) \neq g(1) \wedge c - 3 \wedge c + 1 = d - 4$$

⇩ The function g implies that for all x and y,
if x = y, then g (x) = g (y) (*congruence rule*).

$$step\ 5: SAT\ (\mathrm{c} = 5, \mathrm{d} = 10)$$

- SMT solvers also apply:
  - standard algebraic reduction rules $r \wedge false \mapsto false$
  - contextual simplification $a = 7 \wedge p(a) \mapsto a = 7 \wedge p(7)$

Modeling with non-determinism

Extend C with three modelling features:

- **assert(e)**: aborts execution when e is false, no-op otherwise

```
void assert (_Bool e) { if (!e)  exit(); }
```

- **nondet_int()**: returns non-deterministic int-value

```
int nondet_int () { int x; return x; }
```

- **assume(e)**: “ignores” execution when e is false, no-op otherwise

```
void assume (_Bool e) { while (!e) ;  }
```

General Approach

- Use a C program to set up the structure and deterministic computations
- Use non-determinism to set up search space
- Use assumptions to constrain search space
- Use failing assertion to start the search

```
int main() {
  int x=nondet_int(),y=nondet_int(),z=nondet_int();
  __ESBMC_assume(x > 0 && y > 0 && z > 0);
  __ESBMC_assume(x < 16384 && y < 16384 && z < 16384);
  assert(x*x + y*y != z*z);
  return 0;
}
```

Agenda

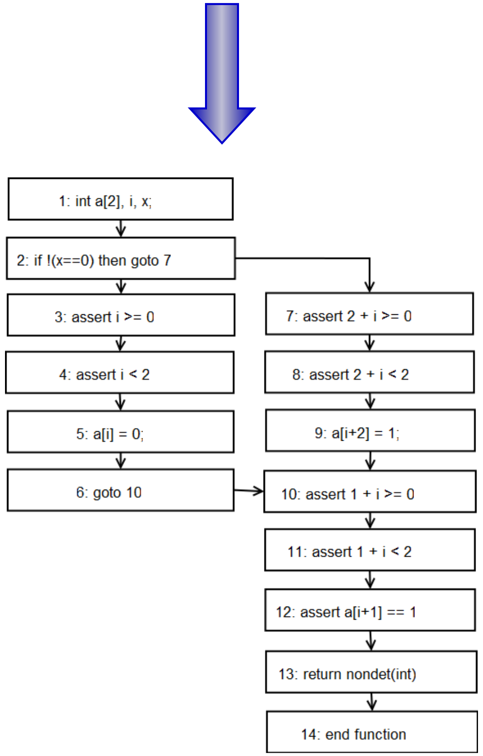
- Intoduce typical BMC Architectures for Verifying Software Systems
- Software Verification and Testing with the ESBMC Framework
- Towards Self-Healing Software via Large Language Models and Formal Verification
- Towards Verification of Programs for CHERI Platforms with ESBMC

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”);
}
```

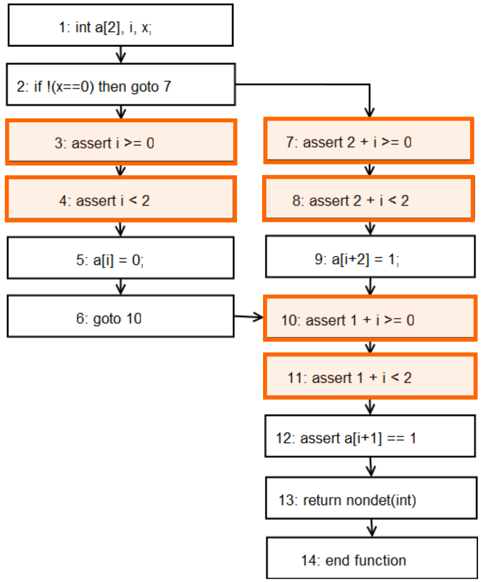


Software BMC

- 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”);
}
```

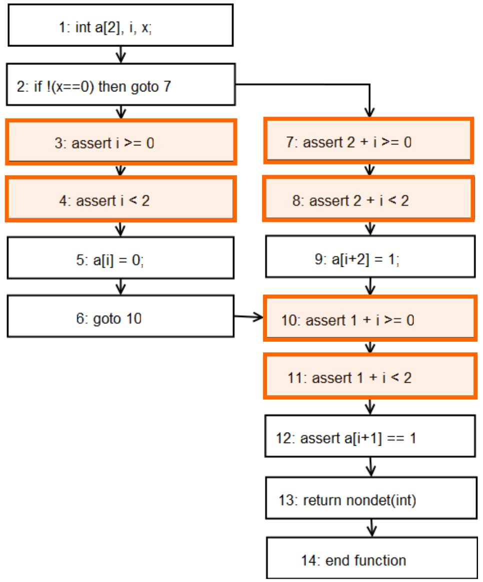


Software BMC

- 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”);
}
```



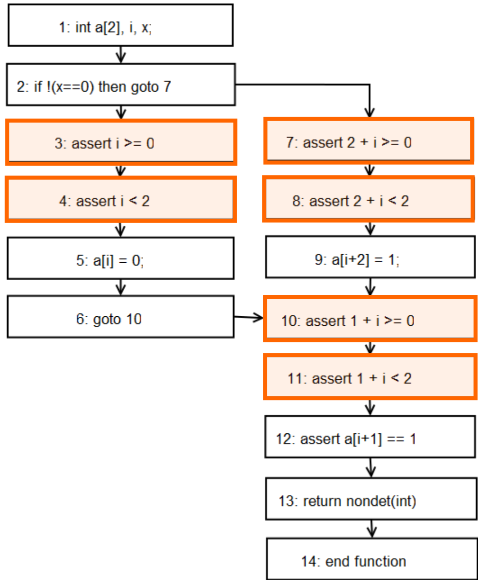
Software BMC

- 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/slicing
  - forward substitutions/caching
  - unreachable code/pointer analysis

} 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/slicing
  - forward substitutions/caching
  - unreachable code/pointer analysis

  } 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");
}
```

⇓

$$
\begin{aligned}
g_1 &= x_1 == 0 \\
a_1 &= a_0 \text{ WITH } [i_0:=0] \\
a_2 &= a_0 \\
a_3 &= a_2 \text{ WITH } [2+i_0:=1] \\
a_4 &= g_1 \,?\, a_1 : a_3 \\
t_1 &= a_4[1+i_0] == 1
\end{aligned}
$$

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/slicing
  - forward substitutions/caching
  - unreachable code/pointer analysis

  } 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

- 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/slicing
  - forward substitutions/caching
  - unreachable code/pointer analysis

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

```
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

- 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/slicing
  - forward substitutions/caching
  - unreachable code/pointer analysis

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

```
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");
}
```

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

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

Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-Based Bounded Model Checking for Embedded ANSI-C Software. IEEE Trans. Software Eng. 38(4): 957-974 (2012)

Induction-Based Verification for Software

***k*-induction** checks loop-free programs...

- **base case (*base$_k$*):** find a counter-example with up to *k* loop unwindings (plain BMC)
- **forward condition (*fwd$_k$*):** check that *P* holds in all states reachable within *k* unwindings
- **inductive step (*step$_k$*):** check that whenever *P* holds for *k* unwindings, it also holds after next unwinding
  - havoc variables
  - assume loop condition
  - run loop body (*k* times)
  - assume loop termination

⇒ iterative deepening if inconclusive

Gadelha, M., Ismail, H., Cordeiro, L.: Handling loops in bounded model checking of C programs via k-induction. Int. J. Softw. Tools Technol. Transf. 19(1): 97-114 (2017)

Induction-Based Verification for Software

k=1
while *k<=max_iterations* **do**
    if $base_{P,\phi,k}$ **then**
        return *trace s[0..k]*
    else
        k=k+1
        if $fwd_{P,\phi,k}$ **then**
            return *true*
        else if $step_{P',\phi,k}$ **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);
```

Automatic Invariant Generation

- Infer invariants based on **intervals** as abstract domain via a dependence graph
  - *E.g., a ≤ x ≤ b (integer and floating-point)*
  - Inject intervals as assumptions and contract them via CSP
  - Remove unreachable states

Line	*Interval for "a"*	*Restriction*
4	$(-\infty, +\infty)$	*None*
6	$(-\infty, 100]$	$a \leq 100$
7	$(100, +\infty)$	$a > 100$

```
1 int main()
2 {
3   int a = *;
4
5   while(a <= 100)
6     a++;
7   assert(a>10);
8   return 0;
9 }
```

k-Induction proof rule "hijacks" loop conditions to nondeterministic values, thus computing intervals become essential

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

Gadelha, M., Monteiro, F., Cordeiro, L., Nicole, D.: ESBMC v6.0: Verifying C Programs Using k-Induction and Invariant Inference - (Competition Contribution). TACAS (3) 2019: 209-213

Computing Intervals

- In ESBMC, the interval has:
  - Lower: represents the lower bound of the interval (or infinity)
  - Upper: represents the upper bound of the interval (or infinity)
  - Lower is always less or equal than upper

- Restrictions are computed through intersection:

$$(-\infty, \infty) \cap (-\infty, 50) = (-\infty, 50)$$

$$(-\infty, \infty) \cap [50, \infty) = [50, \infty)$$

- Merging is computed with the Hull operation:

$$[3,3] \sqcup [5,5] = [3,5]$$

```c
int main()
{
  int a;
  if(a < 50) {
    // ...
    a = 3;
  }
  else {
    // ...
    a = 5;
  }
  // ...
}
```

Computing Intervals

```c
int main()
{
   int a;
   if(a < 50) {
     // ...
     a = 3;
   }
   else {
     // ...
     a = 5;
   }
   // ...
}
```

```c
int main()
{
   int a;
   if(a < 50) {
     __ESBMC_assume(a < 50);
     // ...
     a = 3;
   }
   else {
     __ESBMC_assume(a >= 50);
     // ...
     a = 5;
   }
   __ESBMC_assume(a >= 3 && a <= 5);
   // ...
}
```

BMC of Software Using Interval Methods via Contractors

1) Analyze intervals and properties
   - Static Analysis / Abstract Interpretation
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

```
unsigned int x=nondet_uint();
unsigned int y=nondet_uint();
__ESBMC_assume(x >= 20 && x <= 30);
__ESBMC_assume(y <= 30);
assert(x >= y);
```

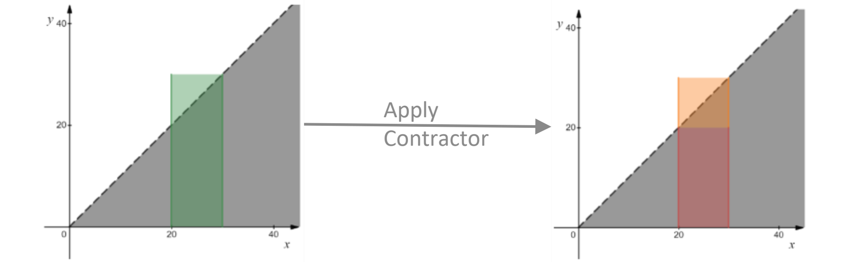
```
__ESBMC_assume(y <= 30 && y >= 20);
```

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

```
unsigned int x=nondet_uint();
unsigned int y=nondet_uint();
__ESBMC_assume(x >= 20 && x <= 30);
__ESBMC_assume(y <= 30);
assert(x >= y);
```

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

Constraint: $y - x \leq 0$

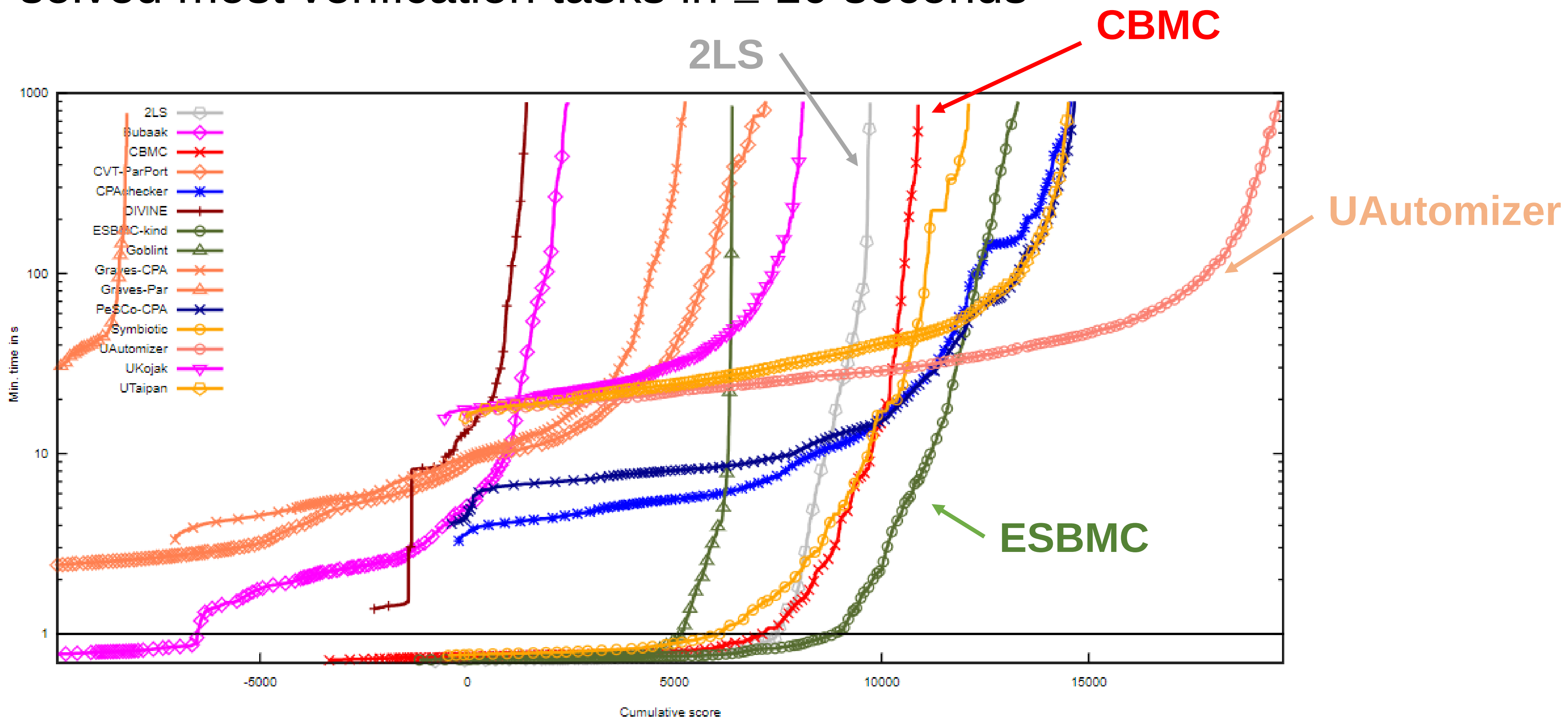


$[x] = [20, 30]$ and $[y] = [0, 30]$ $\quad$ $[x] = [20, 30]$ and $[y] = [20, 30]$

$f(x) > 0$	$I = [0, \infty)$	
$f(x) = y - x$	$[f(x)_1] = I \cap [y_0] - [x_0]$	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

Intl. Software Verification Competition (SV-Comp 2023)

- SV-COMP 2023, 23805 verification tasks, max. score: 38644
- ESBMC solved most verification tasks in ≤ 10 seconds

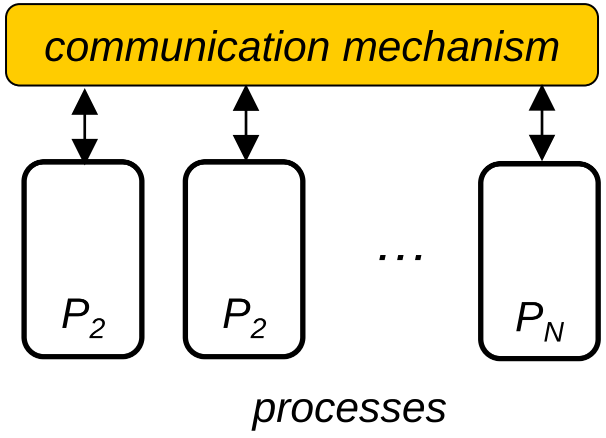


Verification of the `Overall` Category

Concurrency verification

Writing concurrent programs is DIFFICULT

- programmers have to guarantee
  - correctness of sequential execution of each individual process
  - with nondeterministic interferences from other processes (schedules)



- rare schedules result in errors that are difficult to find, reproduce, and repair
  - testers can spend weeks chasing a single bug

⇒ ***huge productivity problem***

Concurrency Errors

There are two main kinds of concurrency errors:

- progress errors: deadlock, starvation, ...
  - typically caused by wrong synchronization
  - requires modeling of synchronization primitives
    - mutex locking / unlocking
  - requires modeling of (global) error condition
- safety errors: assertion violation, ...
  - typically caused by data races (i.e., unsynchronized access to shared data)
  - requires modeling of synchronization primitives
  - can be checked locally

⇒ focus here on safety errors

Shared memory concurrent programs

Concurrent programming styles:

- communication via message passing
  - "truly" parallel distributed systems
  - multiple computations advancing simultaneously
- communication via shared memory
  - multi-threaded programs
  - only one thread active at any given time (conceptually), but active thread can be changed at any given time
    - active == uncontested access to shared memory
    - can be single-core or multi-core

⇒ focus here on multi-threaded, shared memory programs

Multi-threaded programs

- typical C-implementation: `pthreads`
- formed of individual sequential programs (threads)
  - can be created and destroyed on the fly
  - typically for BMC: assume upper bound
  - each possibly with loops and recursive function calls
  - each with local variables
- each thread can read and write shared variables
  - assume sequential consistency: writes are immediately visible to all the other programs
  - weak memory models can be modeled
- execution is interleaving of thread executions
  - only valid for sequential consistency

Concurrency Verification Approaches

- Explicit schedule exploration (ESBMC)
  - lazy exploration
  - schedule recording
- Partial order methods (CBMC)
- Sequentialization
  - KISS
  - Lal / Reps (eager sequentialization)
  - Lazy CSeq
  - memory unwinding

Context-Bounded Model Checking in ESBMC

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

… combines

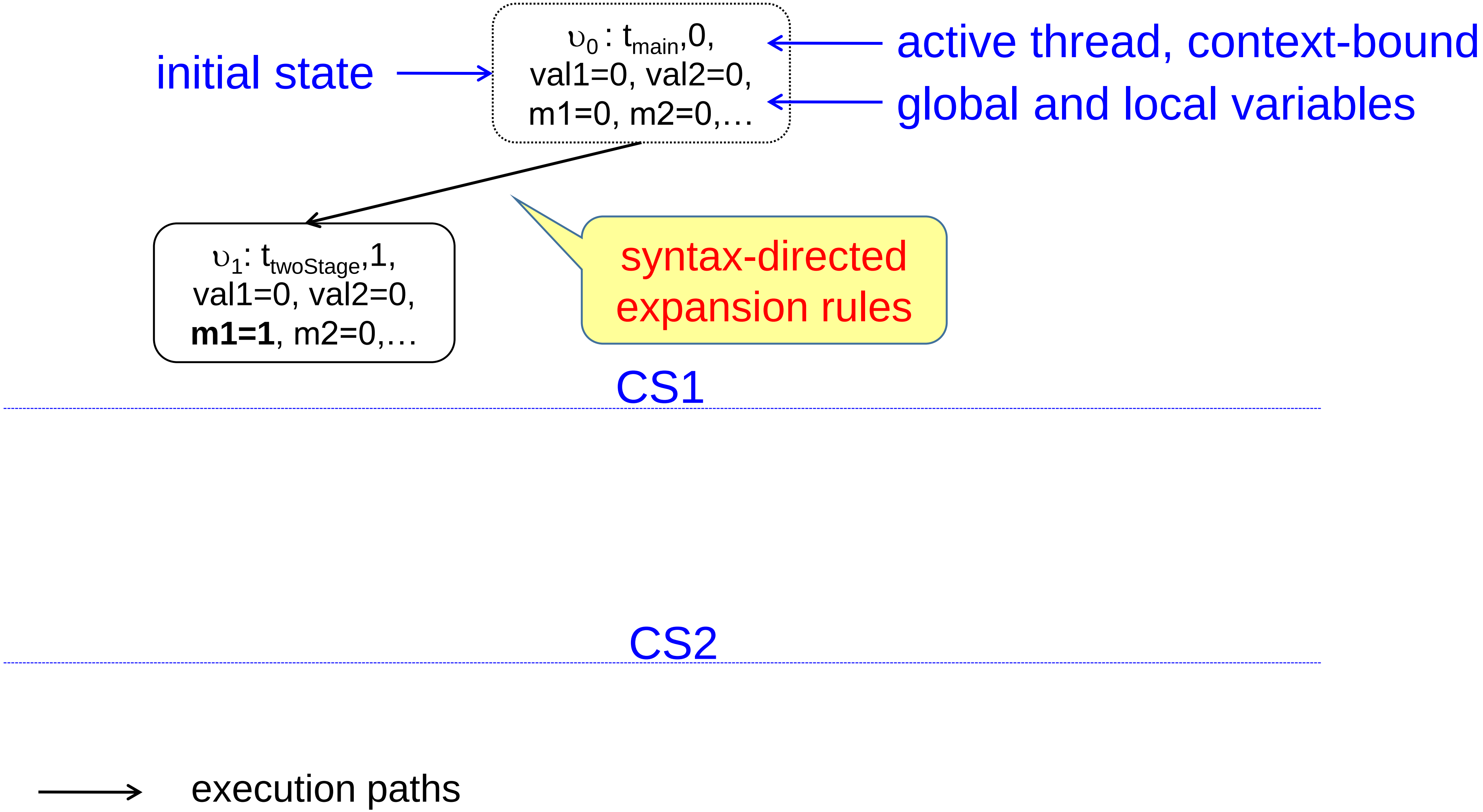
- **symbolic** model checking: on each individual interleaving
- **explicit state** model checking: explore all interleavings
  - bound the number of context switches allowed among threads

… implements

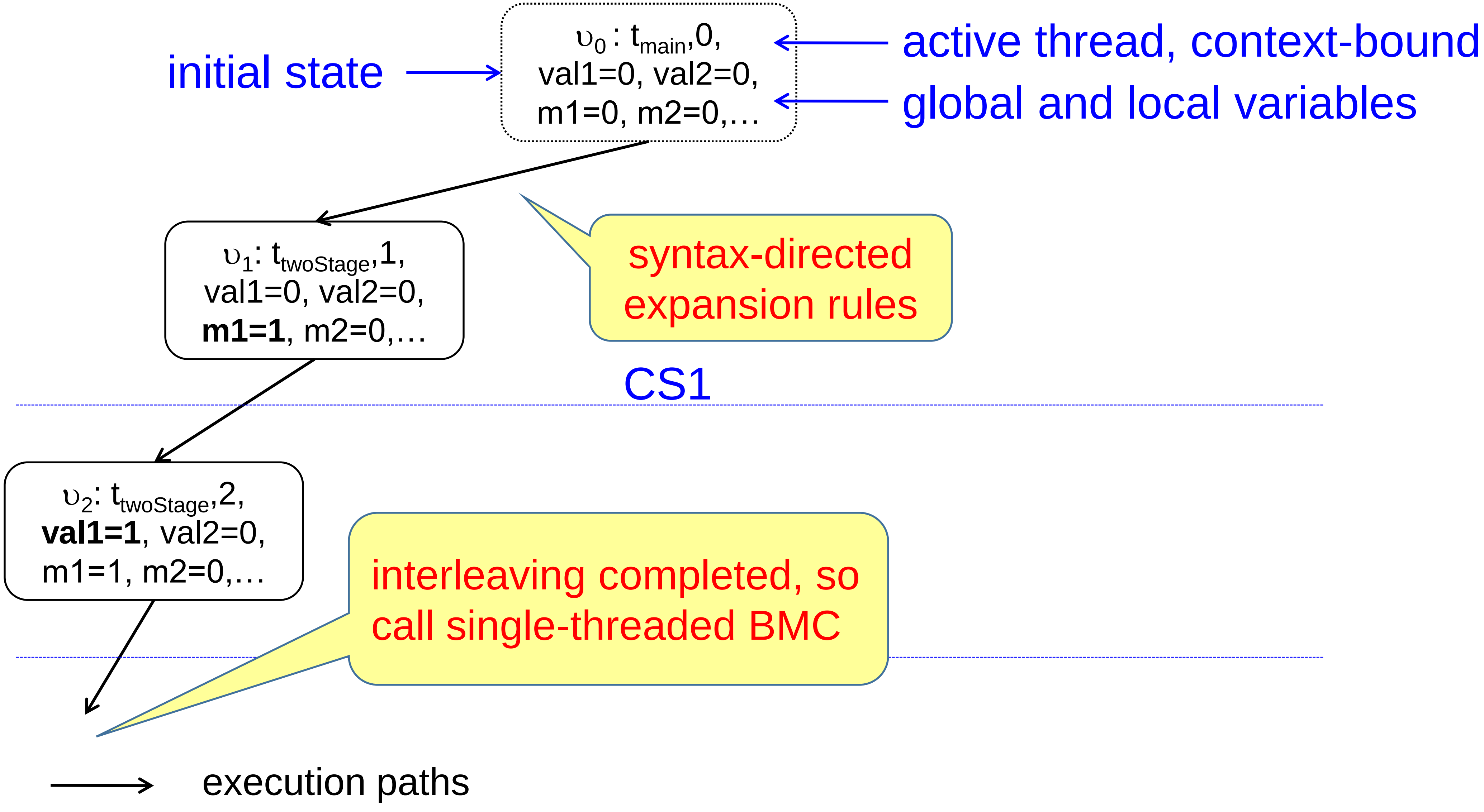
- **symbolic state hashing** (SHA1 hashes)
  - **monotonic partial order** reduction that combines dynamic POR with symbolic state space exploration

Lucas C. Cordeiro, Bernd Fischer: Verifying multi-threaded software using smt-based context-bounded model checking. ICSE 2011: 331-340

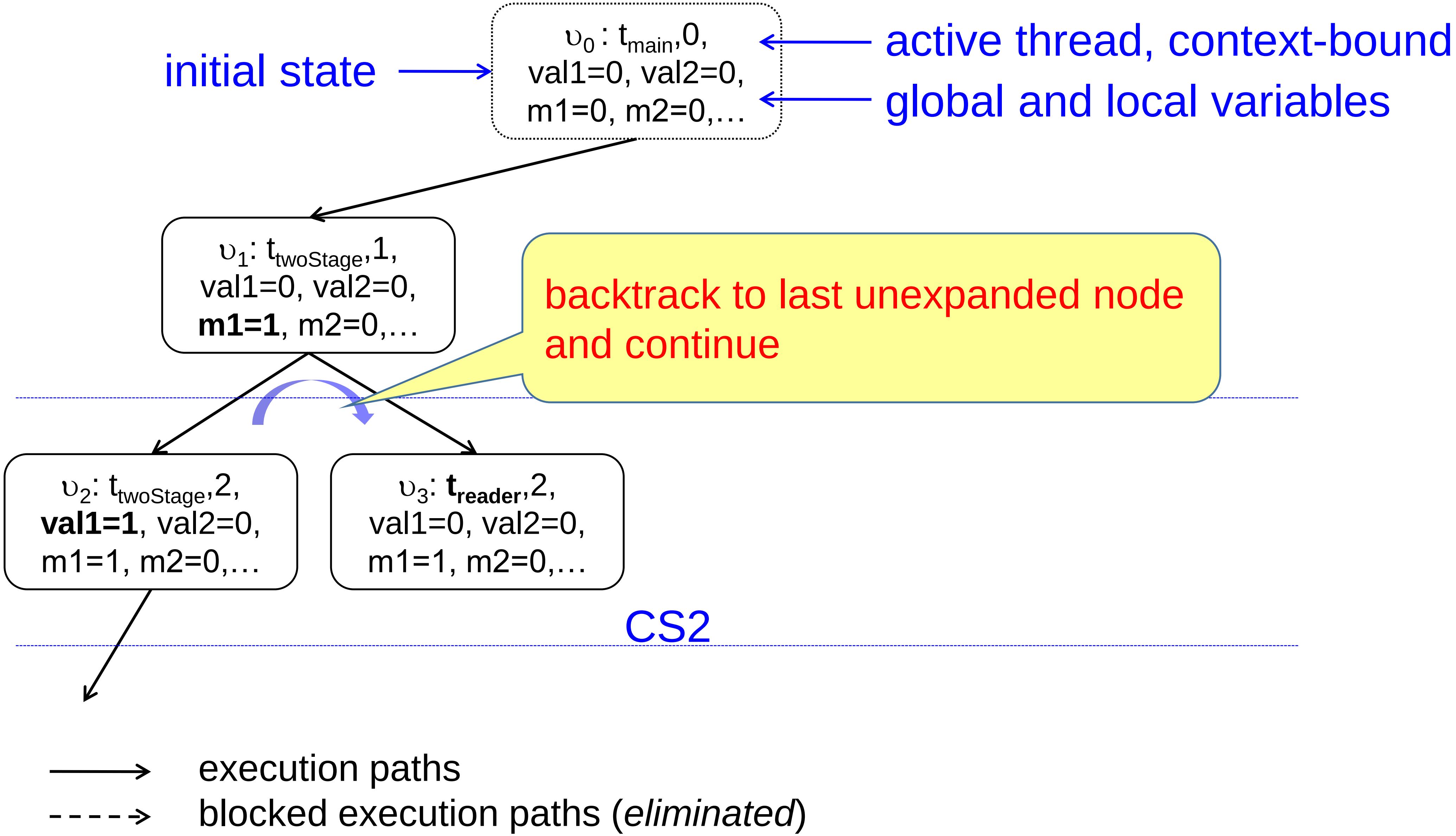
Lazy Exploration of the Reachability Tree



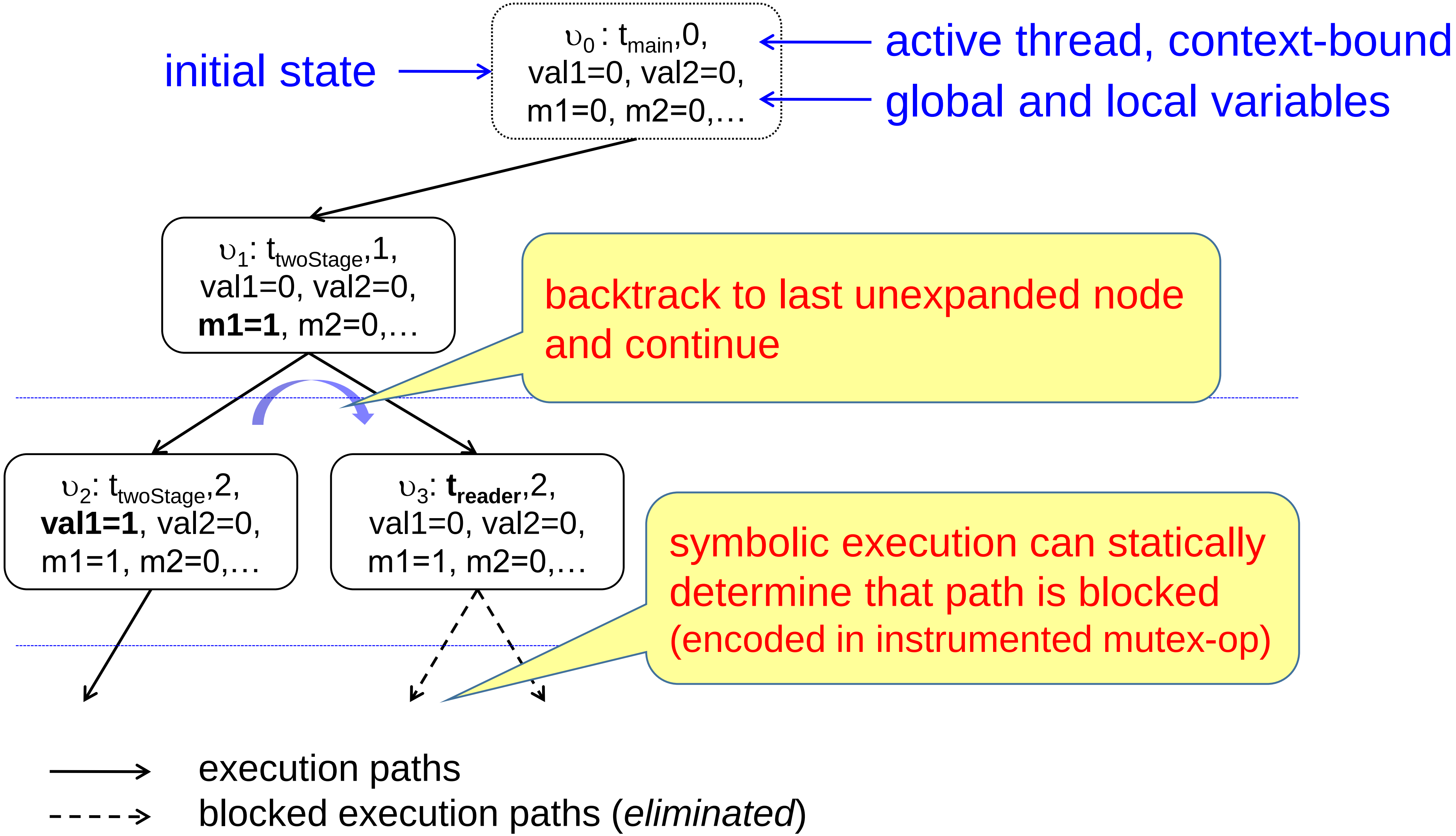
Lazy Exploration of the Reachability Tree



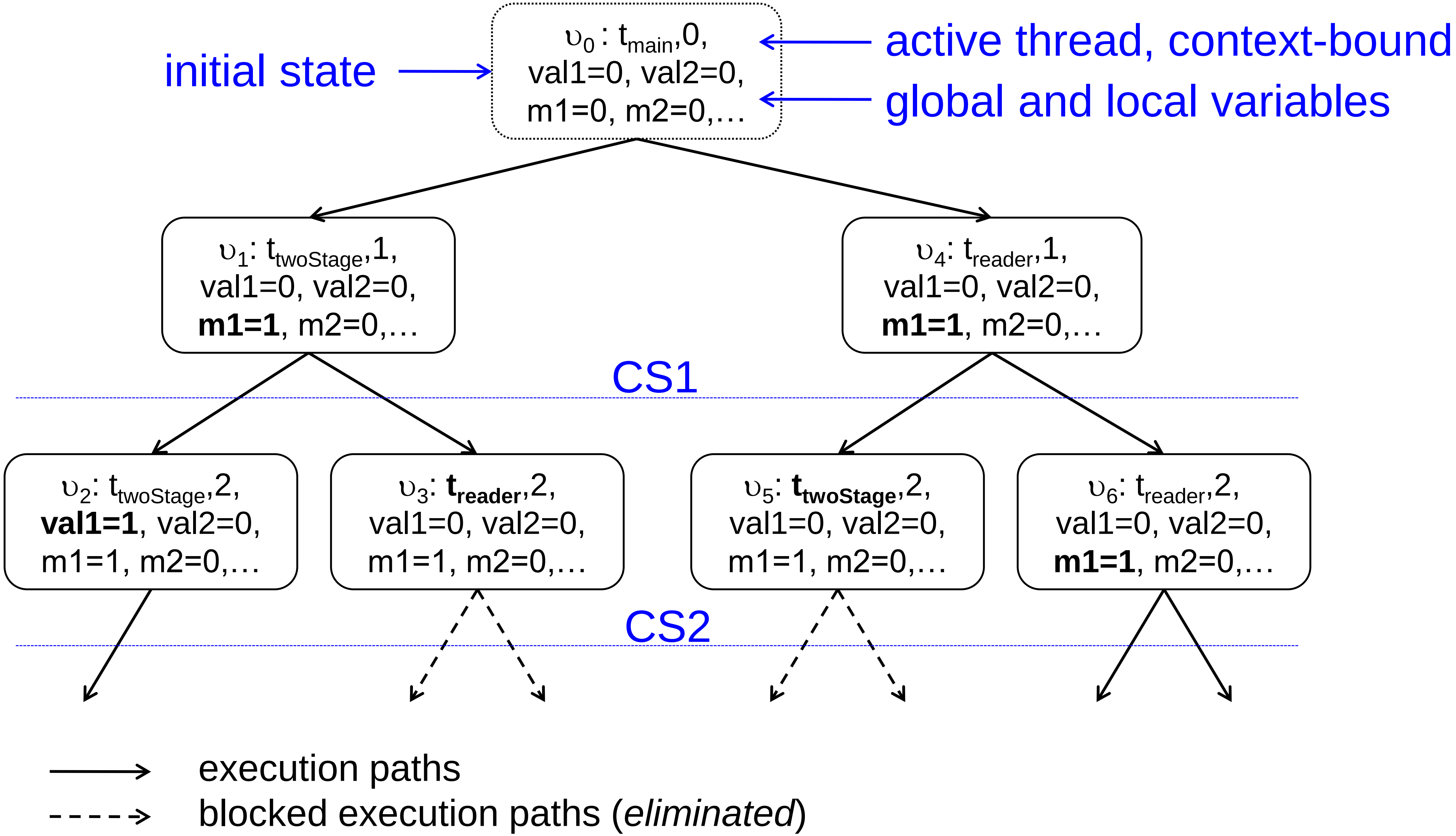
Lazy Exploration of the Reachability Tree



Lazy Exploration of the Reachability Tree



Lazy Exploration of the Reachability Tree



Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

Thread twoStage
```
1:  lock(m1);
2:  val1 = 1;
3:  unlock(m1);
4:  lock(m2);
5:  val2 = val1 + 1;
6:  unlock(m2);
```

```
= 0) {
      (m1);
}
15: unlock(m2);
16: assert(t2==(t1+1));
```

program counter: 0
mutexes: m1 = 0   m2 = 0
globals: val1 = 0   val2 = 0
locals: t1 = 0   t2 = 0

val1 and val2 should be updated synchronously

program state;
(value of program counter and program variables)

Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

interleaving #1: 1

```
Thread twoStage
1:  lock(m1);
2:  val1 = 1;
3:  unlock(m1);
4:  lock(m2);
5:  val2 = val1 + 1;
6:  unlock(m2);
```

program counter: 1
mutexes: ***m1 = 1*** *m2 = 0*
globals: val1 = 0 val2 = 0
locals: t1 = 0 t2 = 0

```
Thread reader
7:   lock(m1);
8:   if (val1 == 0) {
9:      unlock(m1);
10:     return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

interleaving #1: 1-2

```
Thread twoStage
1:  lock(m1);
2:  val1 = 1;
3:  unlock(m1);
4:  lock(m2);
5:  val2 = val1 + 1;
6:  unlock(m2);
```

```
Thread reader
7:   lock(m1);
8:   if (val1 == 0) {
9:       unlock(m1);
10:      return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

program counter: 2
mutexes: m1 = 1 m2 = 0
*globals: **val1 = 1** val2 = 0*
locals: t1 = 0 t2 = 0

Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

interleaving #1: 1-2-3

```
Thread twoStage
1:  lock(m1);
2:  val1 = 1;
3:  unlock(m1);
4:  lock(m2);
5:  val2 = val1 + 1;
6:  unlock(m2);
```

program counter: 3
*mutexes: **m1 = 0** m2 = 0*
globals: val1 = 1 val2 = 0
locals: t1 = 0 t2 = 0

```
Thread reader
7:   lock(m1);
8:   if (val1 == 0) {
9:      unlock(m1);
10:     return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

interleaving #1: 1-2-3–7

Thread twoStage

```
1:  lock(m1);
2:  val1 = 1;
3:  unlock(m1);
4:  lock(m2);
5:  val2 = val1 + 1;
6:  unlock(m2);
```

CS1

Thread reader

```
7:   lock(m1);
8:   if (val1 == 0) {
9:       unlock(m1);
10:      return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

program counter: 7
*mutexes: **m1 = 1** m2 = 0*
globals: val1 = 1 val2 = 0
locals: t1 = 0 t2 = 0

Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

interleaving #1: 1-2-3–7-8

Thread twoStage

```
1:  lock(m1);
2:  val1 = 1;
3:  unlock(m1);
4:  lock(m2);
5:  val2 = val1 + 1;
6:  unlock(m2);
```

CS1

Thread reader

```
7:   lock(m1);
8:   if (val1 == 0) {
9:       unlock(m1);
10:      return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

program counter: 8
mutexes: m1 = 1 m2 = 0
globals: val1 = 1 val2 = 0
locals: t1 = 0 t2 = 0

Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

interleaving #1: 1-2-3–7-8-11

Thread twoStage
```
1:  lock(m1);
2:  val1 = 1;
3:  unlock(m1);
4:  lock(m2);
5:  val2 = val1 + 1;
6:  unlock(m2);
```

CS1

Thread reader
```
7:   lock(m1);
8:   if (val1 == 0) {
9:      unlock(m1);
10:     return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

program counter: 11
mutexes: m1 = 1 m2 = 0
globals: val1 = 1 val2 = 0
*locals: **t1 = 1** t2 = 0*

Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

interleaving #1: 1-2-3–7-8-11-12

```
Thread twoStage
1:  lock(m1);
2:  val1 = 1;
3:  unlock(m1);
4:  lock(m2);
5:  val2 = val1 + 1;
6:  unlock(m2);
```

CS1

```
Thread reader
7:   lock(m1);
8:   if (val1 == 0) {
9:       unlock(m1);
10:      return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

program counter: 12
mutexes: ***m1 = 0*** *m2 = 0*
globals: val1 = 1 val2 = 0
locals: t1 = 1 t2 = 0

Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

interleaving #1: 1-2-3–7-8-11-12–4

Thread twoStage
```
1:  lock(m1);
2:  val1 = 1;
3:  unlock(m1);
4:  lock(m2);
5:  val2 = val1 + 1;
6:  unlock(m2);
```

CS1

CS2

Thread reader
```
7:   lock(m1);
8:   if (val1 == 0) {
9:      unlock(m1);
10:     return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

program counter: 4
mutexes: m1 = 0 ***m2 = 1***
globals: val1 = 1 val2 = 0
locals: t1 = 1 t2 = 0

Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

interleaving #1: 1-2-3–7-8-11-12–4-5

```
Thread twoStage
1:  lock(m1);
2:  val1 = 1;
3:  unlock(m1);
4:  lock(m2);
5:  val2 = val1 + 1;
6:  unlock(m2);
```

CS1

CS2

```
Thread reader
7:   lock(m1);
8:   if (val1 == 0) {
9:      unlock(m1);
10:     return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

program counter: 5
mutexes: m1 = 0   m2 = 1
*globals: val1 = 1   **val2 = 2***
locals: t1 = 1   t2 = 0

Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

interleaving #1: 1-2-3–7-8-11-12–4-5-6

```
Thread twoStage
1:  lock(m1);
2:  val1 = 1;
3:  unlock(m1);
4:  lock(m2);
5:  val2 = val1 + 1;
6:  unlock(m2);
```

CS1

CS2

```
Thread reader
7:   lock(m1);
8:   if (val1 == 0) {
9:      unlock(m1);
10:     return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

program counter: 6
mutexes: m1 = 0 ***m2 = 0***
globals: val1 = 1 val2 = 2
locals: t1 = 1 t2 = 0

Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

interleaving #1: 1-2-3–7-8-11-12–4-5-6–13

```
Thread twoStage
1:  lock(m1);
2:  val1 = 1;
3:  unlock(m1);
4:  lock(m2);
5:  val2 = val1 + 1;
6:  unlock(m2);
```

CS1

CS2

CS3

```
Thread reader
7:   lock(m1);
8:   if (val1 == 0) {
9:       unlock(m1);
10:      return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

program counter: 13
mutexes: m1 = 0 ***m2 = 1***
globals: val1 = 1 val2 = 2
locals: t1 = 1 t2 = 0

Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

interleaving #1: 1-2-3–7-8-11-12–4-5-6–13-14

```
Thread twoStage
1:  lock(m1);
2:  val1 = 1;
3:  unlock(m1);
4:  lock(m2);
5:  val2 = val1 + 1;
6:  unlock(m2);
```

CS1

CS2

CS3

```
Thread reader
7:   lock(m1);
8:   if (val1 == 0) {
9:      unlock(m1);
10:     return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

program counter: 14
mutexes: m1 = 1 m2 = 1
globals: val1 = 1 val2 = 2
locals: t1 = 1 ***t2 = 2***

Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

interleaving #1: 1-2-3–7-8-11-12–4-5-6–13-14-15

```
Thread twoStage
1:  lock(m1);
2:  val1 = 1;
3:  unlock(m1);
4:  lock(m2);
5:  val2 = val1 + 1;
6:  unlock(m2);
```

CS1 CS2 CS3

```
Thread reader
7:   lock(m1);
8:   if (val1 == 0) {
9:      unlock(m1);
10:     return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

program counter: 15
mutexes: m1 = 1 ***m2 = 0***
globals: val1 = 1 val2 = 2
locals: t1 = 1 t2 = 2

Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

interleaving #1: 1-2-3–7-8-11-12–4-5-6–13-14-15-16

```
Thread twoStage
1:  lock(m1);
2:  val1 = 1;
3:  unlock(m1);
4:  lock
5:  val2
6:  unlo
```

CS1

```
Thread reader
7:   lock(m1);
8:   if (val1 == 0) {
9:      unlock(m1);
10
1
1
1
14: t      val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

program counter: 16

interleaving completed, so call single-threaded BMC

...so try next interleaving

QF formula is unsatisfiable, i.e., assertion holds

Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

interleaving #2:

Thread twoStage
```
1:  lock(m1);
2:  val1 = 1;
3:  unlock(m1);
4:  lock(m2);
5:  val2 = val1 + 1;
6:  unlock(m2);
```

program counter: 0
mutexes: m1 = 0   m2 = 0
globals: val1 = 0   val2 = 0
locals: t1 = 0   t2 = 0

Thread reader
```
7:   lock(m1);
8:   if (val1 == 0) {
9:       unlock(m1);
10:      return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

interleaving #2: 1-2-3

```
Thread twoStage
1:  lock(m1);
2:  val1 = 1;
3:  unlock(m1);
4:  lock(m2);
5:  val2 = val1 + 1;
6:  unlock(m2);
```

program counter: 3
mutexes: m1 = 0 m2 = 0
globals: val1 = 1 val2 = 0
locals: t1 = 0 t2 = 0

```
Thread reader
7:   lock(m1);
8:   if (val1 == 0) {
9:      unlock(m1);
10:     return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

interleaving #2: 1-2-3–7

```
Thread twoStage
1:  lock(m1);
2:  val1 = 1;
3:  unlock(m1);
4:  lock(m2);
5:  val2 = val1 + 1;
6:  unlock(m2);
```

CS1

```
Thread reader
7:  lock(m1);
8:  if (val1 == 0) {
9:      unlock(m1);
10:     return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

program counter: 7
mutexes: ***m1 = 1*** *m2 = 0*
globals: val1 = 1 val2 = 0
locals: t1 = 0 t2 = 0

Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

interleaving #2: 1-2-3–7-8-11-12-13-14-15-16

Thread twoStage

```
1:  lock(m1);
2:  val1 = 1;
3:  unlock(m1);
4:  lock(m2);
5:  val2 = val1 + 1;
6:  unlock(m2);
```

CS1 →

Thread reader

```
7:   lock(m1);
8:   if (val1 == 0) {
9:       unlock(m1);
10:      return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

program counter: 16
mutexes: m1 = 0   m2 = 0
globals: val1 = 1   val2 = 0
locals: t1 = 1   t2 = 0

Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

interleaving #2: 1-2-3–7-8-11-12-13-14-15-16–4

```
Thread twoStage
1:  lock(m1);
2:  val1 = 1;
3:  unlock(m1);
4:  lock(m2);
5:  val2 = val1 + 1;
6:  unlock(m2);
```

CS1

CS2

```
Thread reader
7:   lock(m1);
8:   if (val1 == 0) {
9:      unlock(m1);
10:     return NULL; }
11: t1 = val1;
12: unlock(m1);
13: lock(m2);
14: t2 = val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

program counter: 4
mutexes: m1 = 0 ***m2 = 1***
globals: val1 = 1 val2 = 0
locals: t1 = 1 t2 = 0

Lazy exploration of interleavings

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

interleaving #2: 1-2-3–7-8-11-12-13-14-15-16–4-5-6

```
Thread twoStage
1:  lock(m1);
2:  val1 = 1;
3:  unlock(m1);
4:  lock(m2);
5:  val2 = val1 + 1;
6:  unlock(m2);
```

interleaving completed, so call single-threaded BMC (again)

CS2

```
14: t… val2;
15: unlock(m2);
16: assert(t2==(t1+1));
```

...so found a bug for a specific interleaving

QF formula is satisfiable, i.e., assertion fails

Results on SV-COMP'20 Benchmarks with Reachable Error Label

			CBMC 5.4					CBMC 5.28					CBMC 5.28 (K)				
sub-category	files	l.o.c.	pass	miss	error	t.o.	time	pass	miss	error	t.o.	time	pass	miss	error	t.o.	time
ldv-races	8	669	3	5	-	-	0.8	8	-	-	-	0.4	8	-	-	-	0.6
pthread	20	1,906	17	-	2	1	142.4	8	-	11	1	159.6	8	-	11	1	103.5
pthread-atomic	2	182	2	-	-	-	0.7	2	-	-	-	0.8	2	-	-	-	0.7
pthread-c-dac	1	1,347	1	-	-	-	6.4	-	-	1	-	0.1	-	-	1	-	0.1
pthread-complex	4	663	-	1	-	3	818.8	-	-	3	1	250.1	-	-	3	1	250.2
pthread-divine	7	440	1	5	-	1	144.6	6	-	1	-	0.7	6	-	1	-	1.4
pthread-driver-races	4	1,216	3	1	-	-	1.2	4	-	-	-	16.2	4	-	-	-	18.8
pthread-ext	8	253	7	-	-	1	276.6	7	-	-	1	275.0	8	-	-	-	18.4
pthread-lit	3	111	2	-	-	1	333.6	2	-	-	1	333.4	2	-	-	1	333.5
pthread-nondet	3	83	3	-	-	-	172.9	3	-	-	-	204.8	3	-	-	-	281.4
pthread-wmm	754	150,270	754	-	-	-	0.6	754	-	-	-	0.3	754	-	-	-	0.6
Totals	814	157,602	793	12	2	7	19.9	794	-	16	4	10.2	795	-	16	3	6.9

Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, Gennaro Parlato: Bounded Verification of Multi-threaded Programs via Lazy Sequentialization. ACM Trans. Program. Lang. Syst. 44(1): 1:1-1:50 (2022)

Results on SV-COMP'20 Benchmarks with Reachable Error Label

			ESBMC 6.4					SMACK 2.4.0					Yogar-CBMC				
sub-category	files	l.o.c.	pass	miss	error	t.o.	time	pass	miss	error	t.o.	time	pass	miss	error	t.o.	time
ldv-races	8	669	8	-	-	-	0.3	7	1	-	-	35.6	4	-	4	-	2.1
pthread	20	1,906	12	3	2	3	230.5	8	6	-	6	382.2	13	-	7	-	52.1
pthread-atomic	2	182	2	-	-	-	48.7	2	-	-	-	48.1	1	-	1	-	1.5
pthread-c-dac	1	1,347	-	-	-	1	1,004.5	-	1	-	-	3.6	1	-	-	-	3.5
pthread-complex	4	663	1	-	2	1	606.9	1	2	-	1	261.3	-	-	4	-	5.1
pthread-divine	7	440	6	-	1	-	101.3	4	2	-	1	150.9	1	-	6	-	2.8
pthread-driver-races	4	1,216	4	-	-	-	4.3	-	4	-	-	22.8	4	-	-	-	0.8
pthread-ext	8	253	8	-	-	-	0.1	1	7	-	-	4.9	5	-	3	-	0.4
pthread-lit	3	111	2	-	-	1	333.5	1	1	-	1	338.8	3	-	-	-	26.0
pthread-nondet	3	83	3	-	-	-	0.2	1	1	-	1	337.0	3	-	-	-	4.5
pthread-wmm	754	150,270	754	-	-	-	42.3	254	8	-	492	795.2	-	-	754	-	0.7
Totals	814	157,602	800	3	5	6	52.8	279	33	-	502	746.2	35	-	779	-	4.9

Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, Gennaro Parlato: Bounded Verification of Multi-threaded Programs via Lazy Sequentialization. ACM Trans. Program. Lang. Syst. 44(1): 1:1-1:50 (2022)

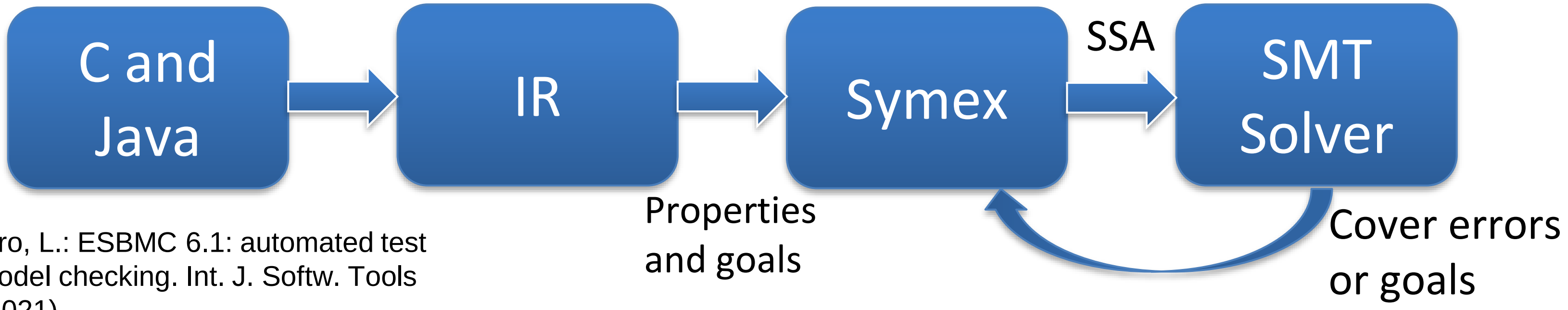
Results on SV-COMP'20 Benchmarks with Reachable Error Label

			CPAchecker 1.9 (CPA-Seq)					Divine 4.4.0					UL-CSeq				
sub-category	files	l.o.c.	pass	miss	error	t.o.	time	pass	miss	error	t.o.	time	pass	miss	error	t.o.	time
ldv-races	8	669	3	-	5	-	4.3	5	3	-	-	2.9	8	-	-	-	146.1
pthread	20	1,906	9	-	11	-	121.8	11	1	8	-	139.9	10	-	1	9	487.2
pthread-atomic	2	182	2	-	-	-	20.0	1	-	1	-	48.2	2	-	-	-	39.0
pthread-c-dac	1	1,347	-	-	1	-	915.6	-	-	-	1	1,000.2	-	-	-	1	1,000.0
pthread-complex	4	663	-	-	4	-	332.8	1	-	1	2	502.1	-	-	-	4	1,000.0
pthread-divine	7	440	2	-	5	-	7.2	4	-	3	-	3.4	3	-	4	-	14.6
pthread-driver-races	4	1,216	-	-	4	-	112.0	-	-	3	1	251.5	-	-	4	-	0.5
pthread-ext	8	253	-	-	8	-	4.0	2	-	5	1	127.5	8	-	-	-	28.7
pthread-lit	3	111	1	-	2	-	3.6	1	-	-	1	335.2	2	-	-	1	334.8
pthread-nondet	3	83	-	-	3	-	3.5	-	1	-	-	2.4	-	3	-	-	340.5
pthread-wmm	754	150,270	626	-	46	82	173.1	548	3	-	4	41.6	754	-	-	-	35.7
Totals	814	157,602	643	-	89	82	168.6	573	210	21	10	49.3	787	3	9	15	55.3

Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, Gennaro Parlato: Bounded Verification of Multi-threaded Programs via Lazy Sequentialization. ACM Trans. Program. Lang. Syst. 44(1): 1:1-1:50 (2022)

White-box Fuzzing: Bug Finding and Code Coverage

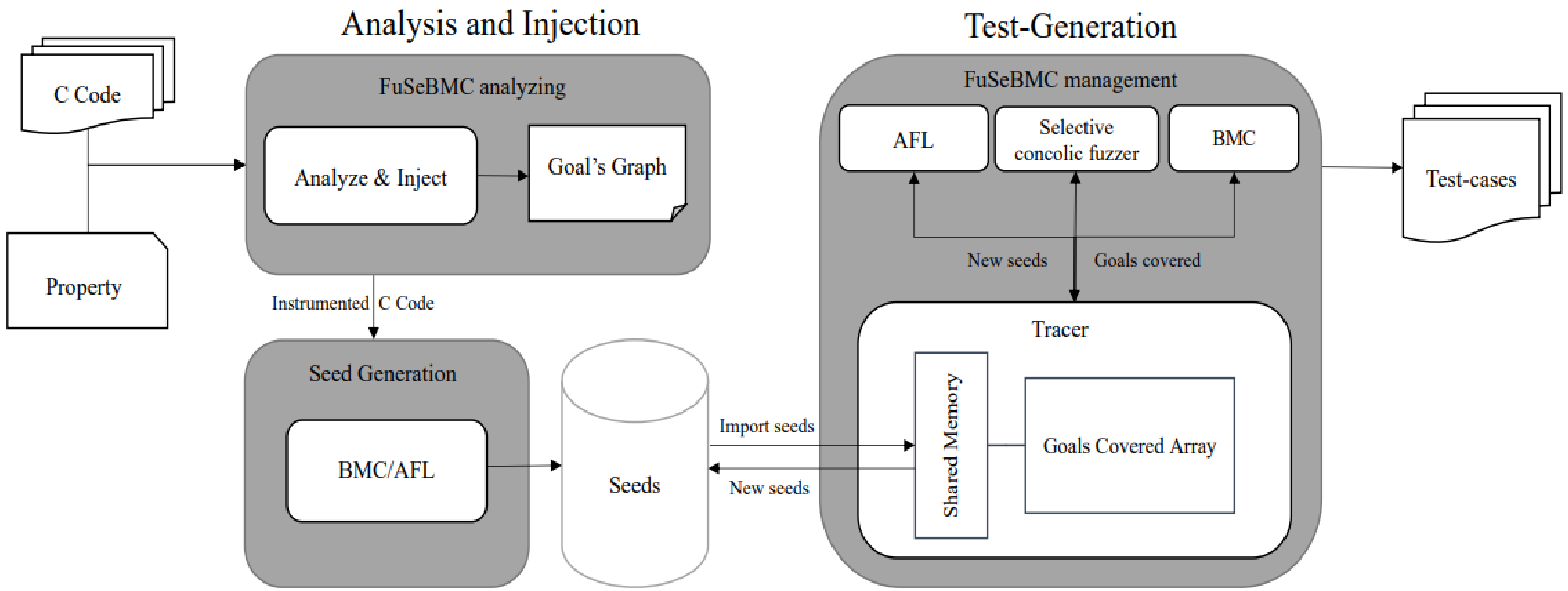
- Translate the program to an **intermediate representation** (IR)
- Add properties to check **errors** or goals to check **coverage**
- **Symbolically** execute IR to produce an SSA program
- Translate the resulting SSA program into a **logical formula**
- Solve the formula iteratively to cover errors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



Gadelha, M., Menezes, R., Cordeiro, L.: ESBMC 6.1: automated test case generation using bounded model checking. Int. J. Softw. Tools Technol. Transf. 23(6): 857-861 (2021).

FuSeBMC v4 Framework

- Use **Clang** tooling infrastructure
- Employ three engines in its **reachability analysis**: **one BMC and two fuzzing engines**
- Use a **tracer** to coordinate the various engines

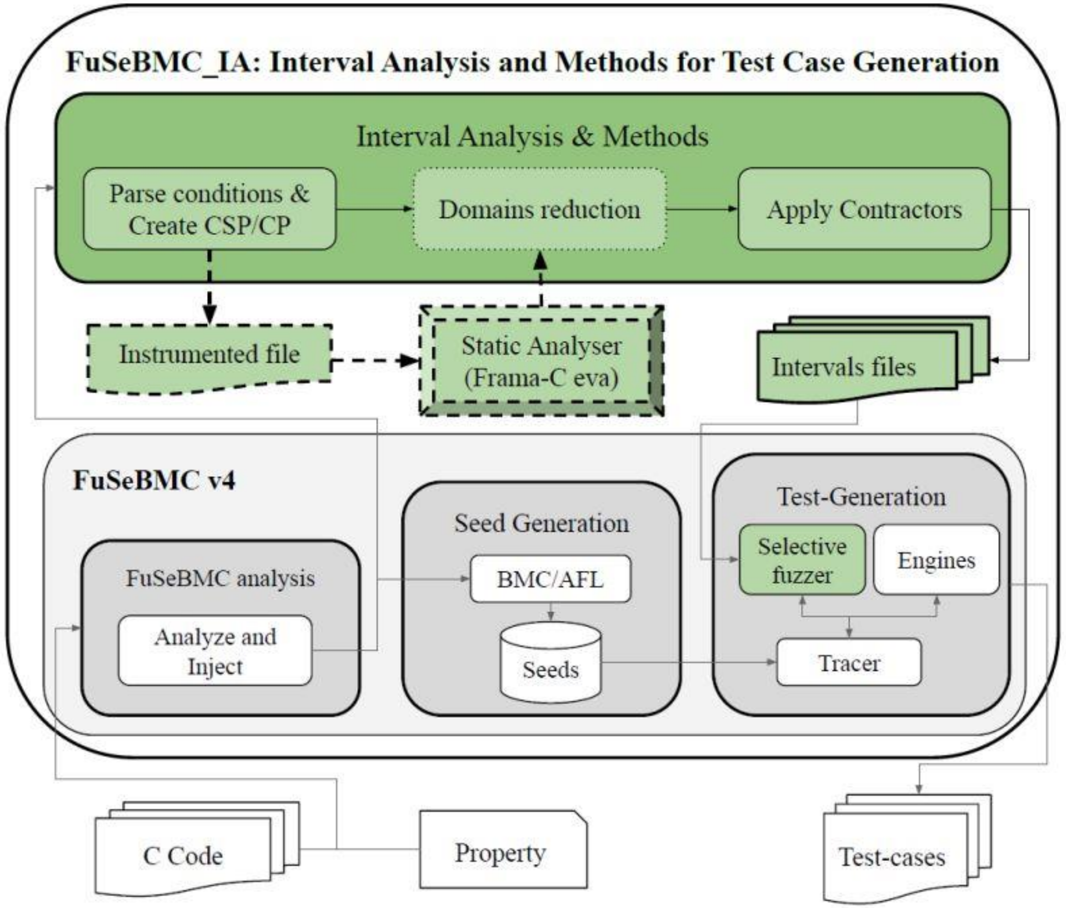


Alshmrany, K., Aldughaim, M., Bhayat, A., Cordeiro, L.: FuSeBMC v4: Smart Seed Generation for Hybrid Fuzzing - (Competition Contribution). FASE 2022: 336-340

Interval Analysis and Methods for Automated Test Case Generation

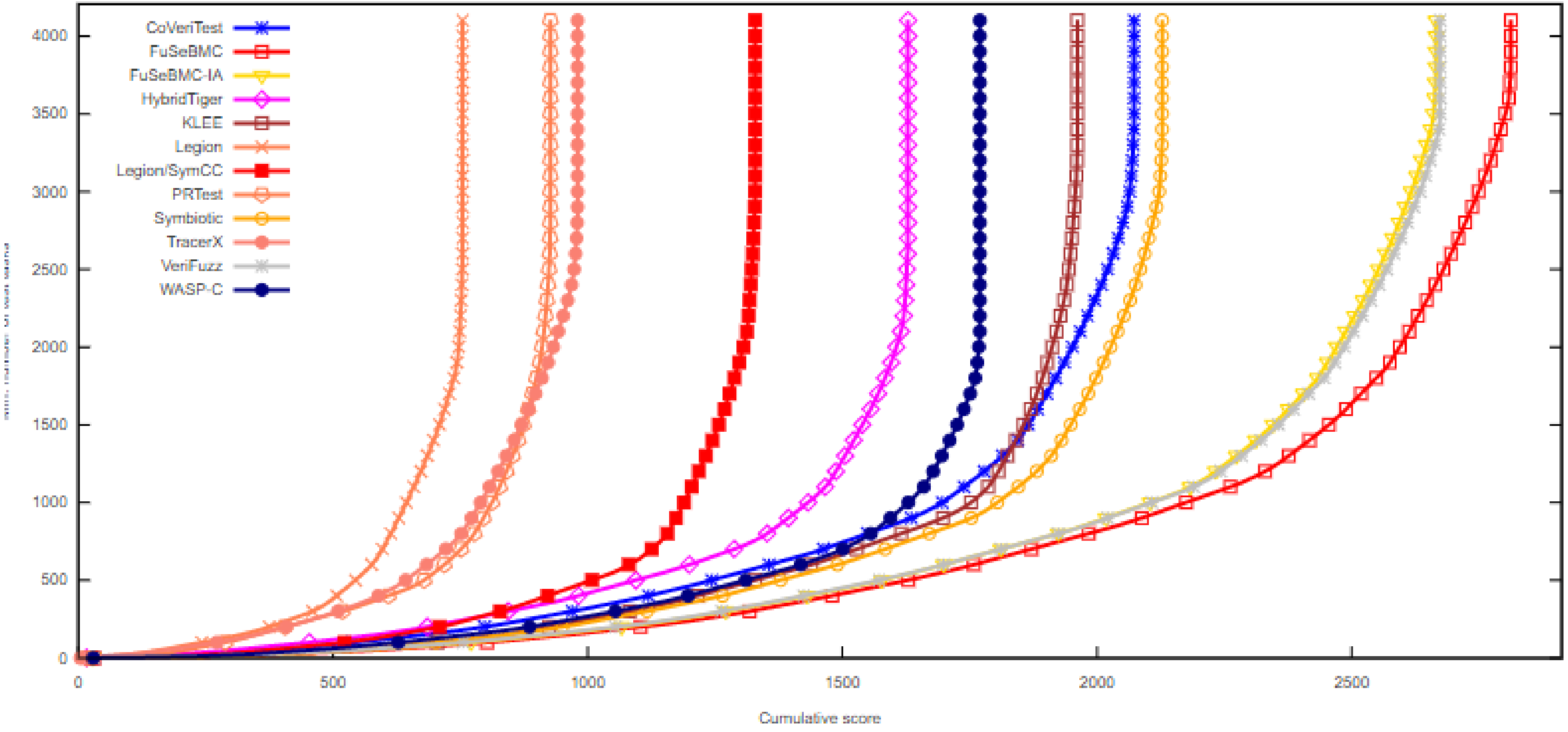
This combined method can **reduce CPU time**, **memory usage**, and **energy consumption**

We advocate that combining **cooperative verification** and **constraint programming** is essential to leverage a **modular cooperative cloud-native testing platform**



Aldughaim, M., Alshmrany, K., Gadelha, M., de Freitas, R., Cordeiro, L.: FuSeBMC_IA: Interval Analysis and Methods for Test Case Generation - (Competition Contribution). FASE 2023: 324-329

Competition on Software Testing 2023: Results of the `Overall` Category

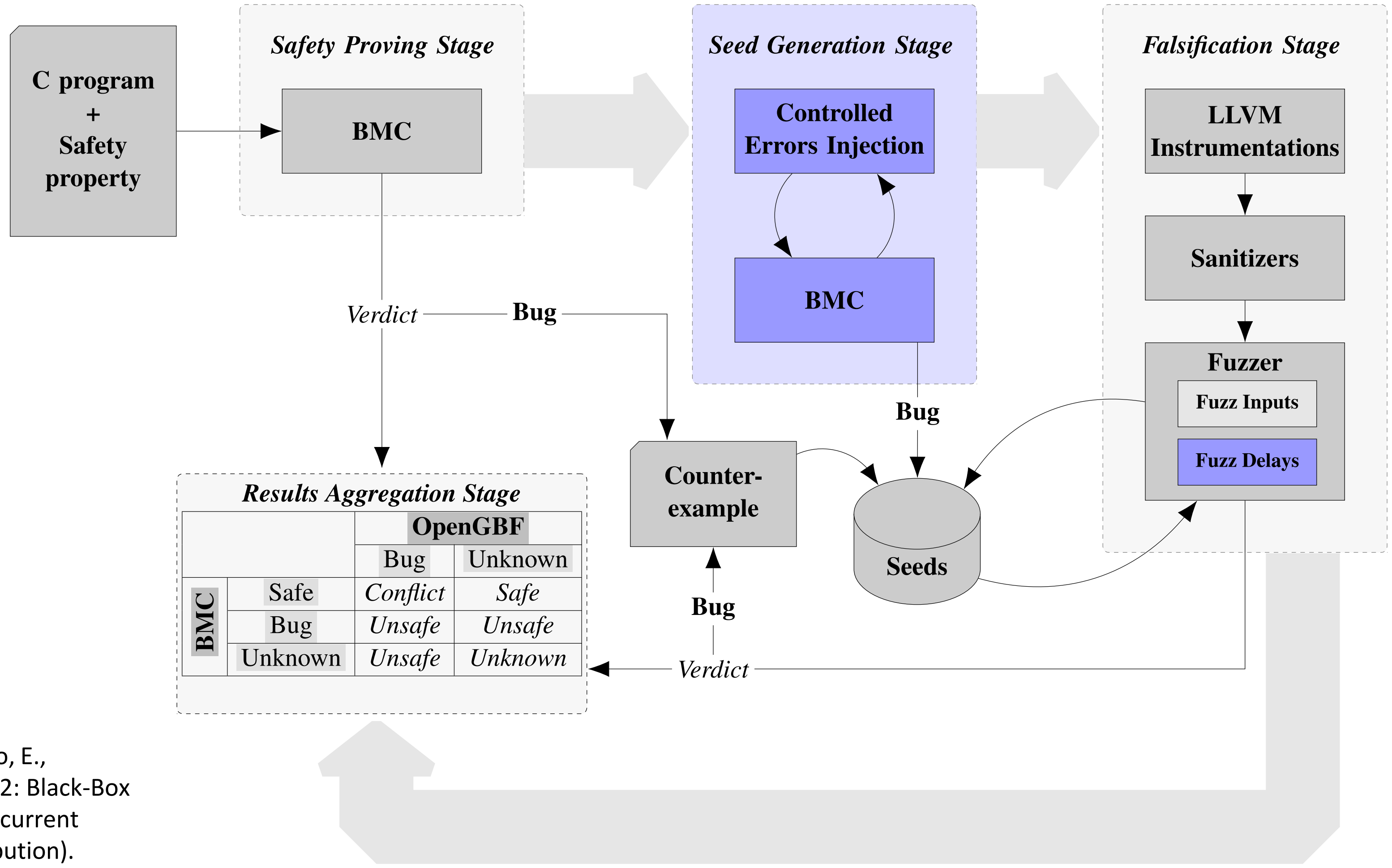


FuSeBMC achieved 3 awards: 1st place in `Cover-Error`, 1st place in `Cover-Branches`, and 1st place in `Overall`

Alshmrany, K., Aldughaim, M., Bhayat, A., Cordeiro, L.: FuSeBMC v4: Smart Seed Generation for Hybrid Fuzzing - (Competition Contribution). FASE 2022: 336-340

https://test-comp.sosy-lab.org/2023/

EBF: Black-Box Cooperative Verification for Concurrent Programs



Aljaafari, F., Shmarov, F., Manino, E., Menezes, R., Cordeiro, L.: EBF 4.2: Black-Box Cooperative Verification for Concurrent Programs - (Competition Contribution). TACAS (2) 2023: 541-546

EBF 4.0 with different BMC tools

- **BMC** 6 min + **OpenGBF** 5 min + **results Aggregation** 4 min = 15 min
- **RAM limit** is 15 GB per Benchexec run
- **ConcurrencySafety main** from SV-COMP 2022
  - Witness validation switched off
- Ubuntu 20.04.4 LTS with 160 GB RAM and 25 cores

Verification outcome	Tool							
	EBF	*Deagle*	*EBF*	*Cseq*	*EBF*	*ESBMC*	*EBF*	*CBMC*
Correct True	240	240	172	**177**	65	**70**	139	**146**
Correct False	**336**	319	**333**	313	**308**	268	**320**	303
Incorrect True	0	0	0	0	0	0	0	0
Incorrect False	0	0	0	0	0	1	0	3
Unknown	**187**	204	**258**	273	**390**	424	**304**	311

- EBF4.0 **increases** the number of **detected bugs for** BMC tools
- EBF4.0 provides a better **trade-off** between **bug finding** and **safety proving** than each BMC engine

WolfMQTT Verification

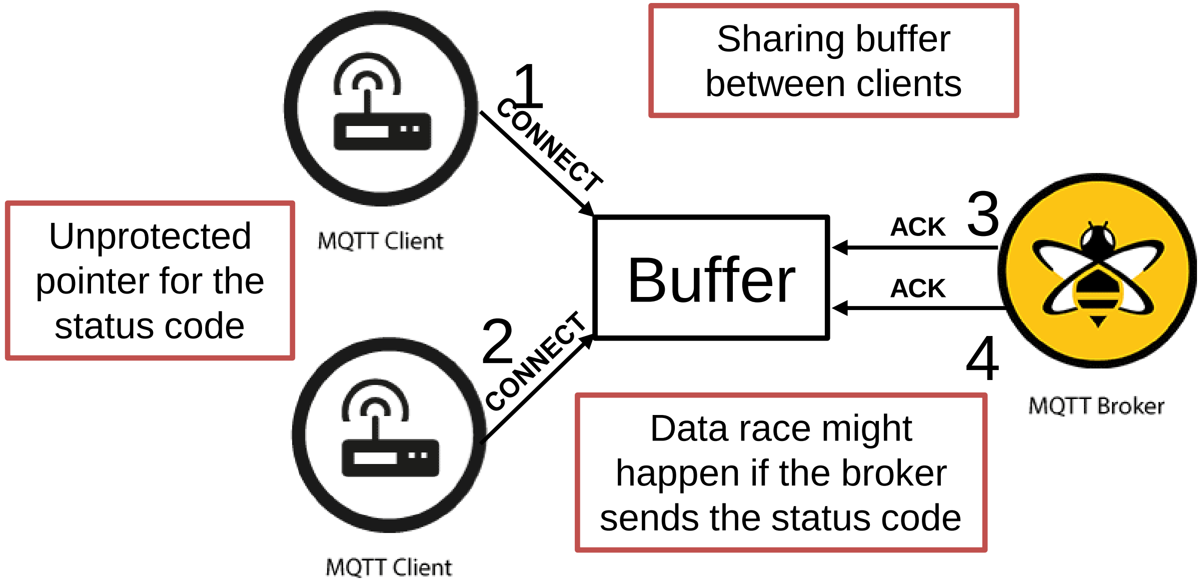
- **wolfMQTT** library is a client implementation of the MQTT protocol written in C for **IoT devices**

`subscribe_task` and `waitMessage_task` are called through different threads accessing `packet_ret`, causing a data race in `MqttClient_WaitType`

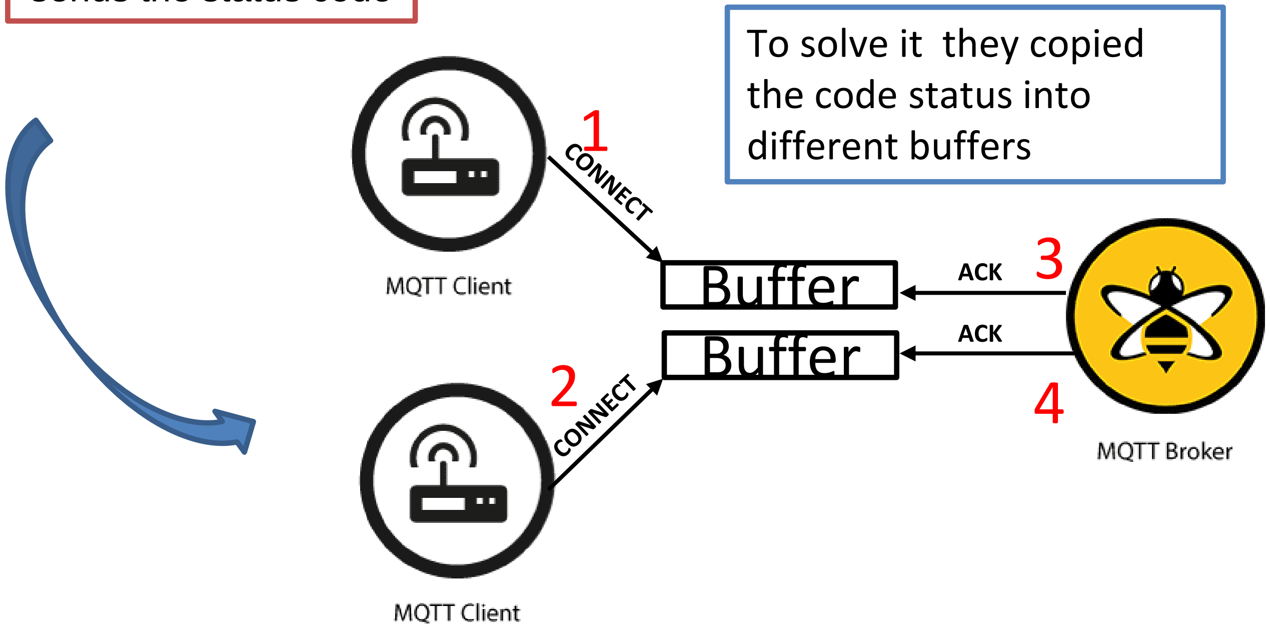
Here is where the data race might happen! Unprotected pointer

```
Int main(){
Pthread_t th1, th2;
static MQTTCtx mqttCtx;
pthread_create(&th1, subscribe_task, &mqttCtx))
pthread_create(&th2, waitMessage_task, &mqttCtx))}

static void *subscribe_task(void *client){
.....
MqttClient_WaitType(client,msg,MQTT_PACKET_TYPE_ANY,
0,timeout_ms);
.....}
static void *waitMessage_task(void *client){
…
MqttClient_WaitType(client, msg, MQTT_PACKET_TYPE_ANY,
0,timeout_ms);
.....}
static int MqttClient_WaitType(MqttClient *client,
void *packet_obj,
   byte wait_type, word16 wait_packet_id, int timeout_ms)
{
.....
           rc = wm_SemLock(&client->lockClient);
           if (rc == 0) {
               if (MqttClient_RespList_Find(client,
(MqttPacketType)wait_type,
                       wait_packet_id, &pendResp)) {
                   if (pendResp->packetDone) {
                       rc = pendResp->packet_ret;
.....}
```

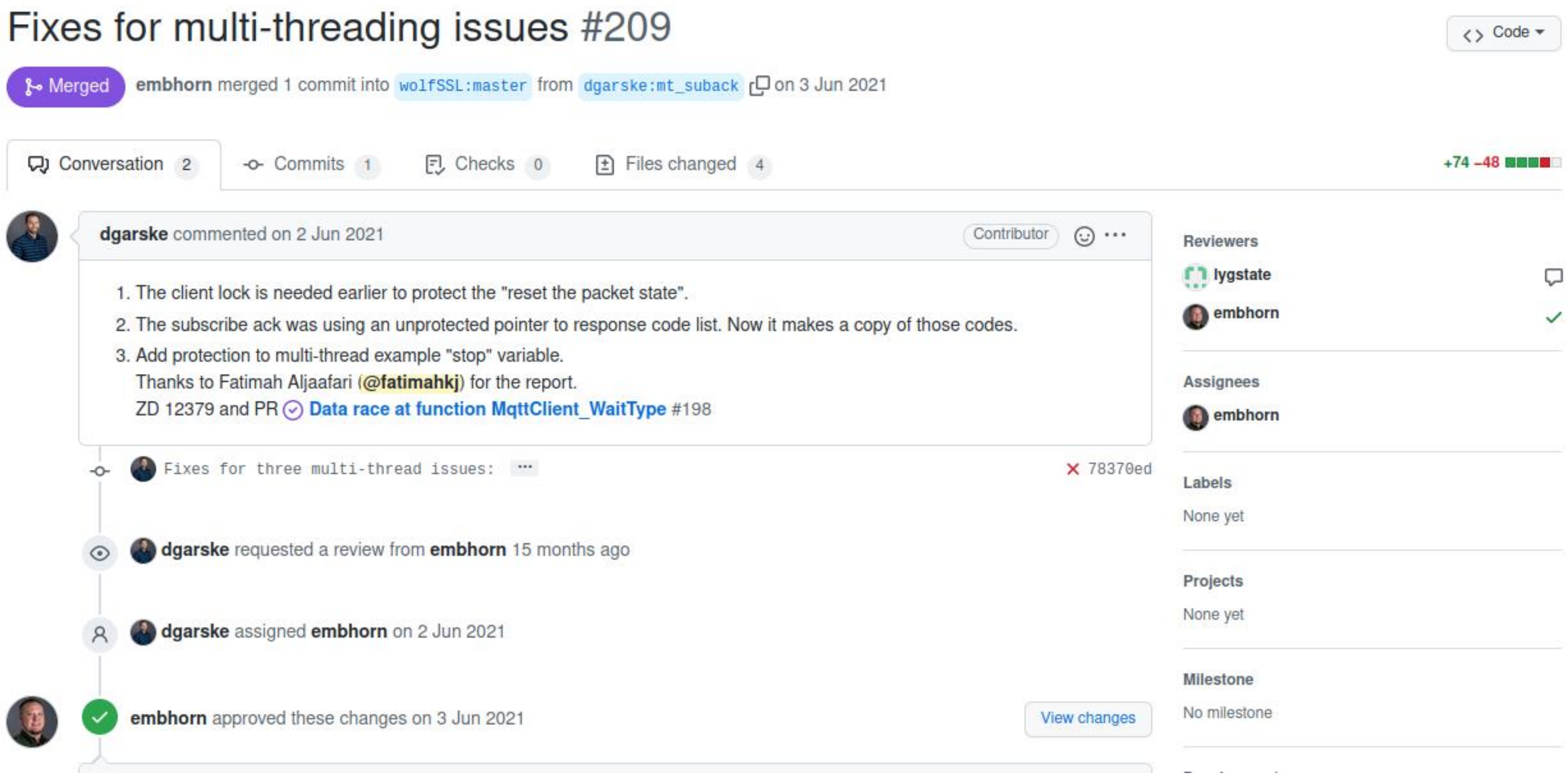
WolfMQTT Verification



After fixing the concurrency vulnerability



Bug Report



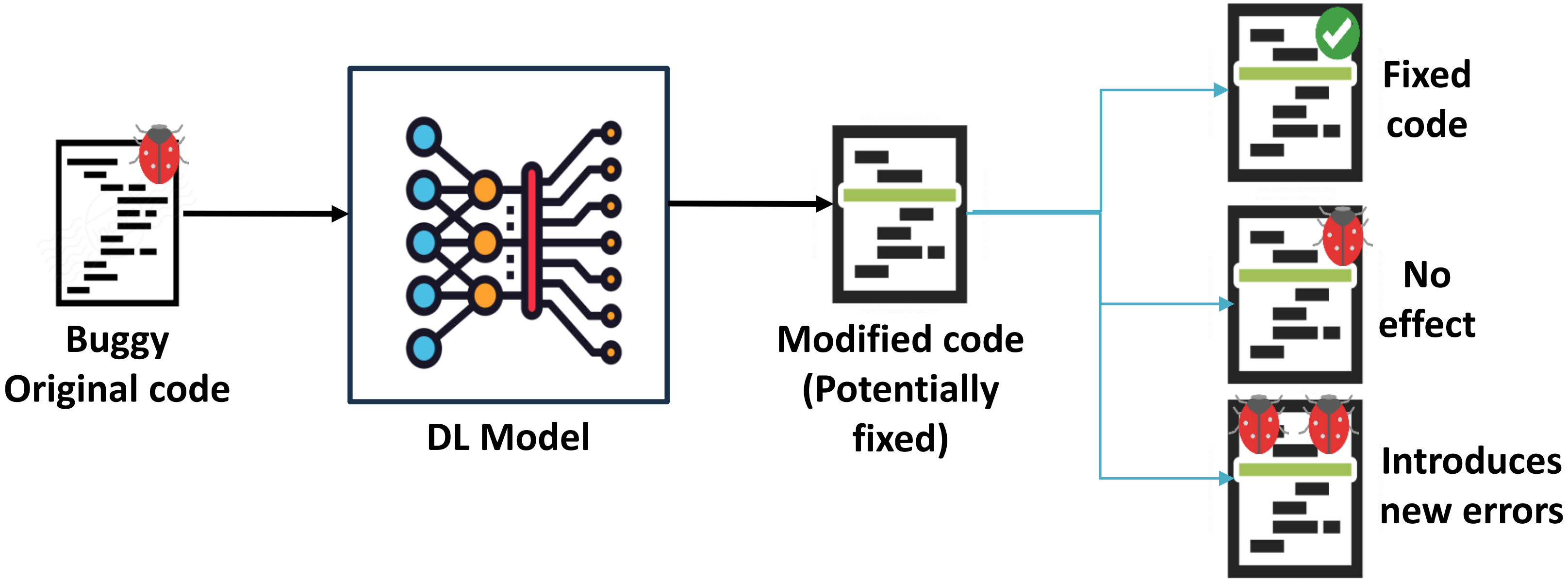
https://github.com/wolfSSL/wolfMQTT


wolfSSL
TLS 1.3
wolfSSL

Agenda

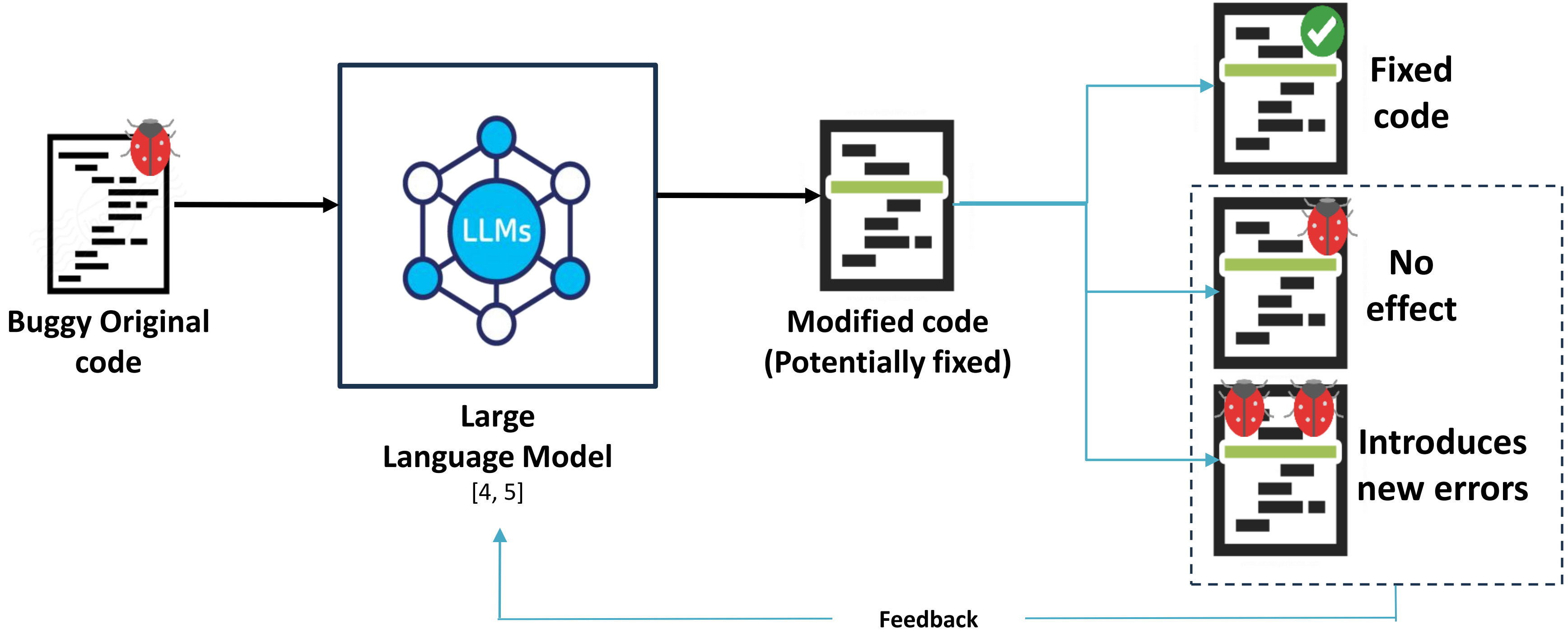
- Intoduce typical BMC Architectures for Verifying Software Systems
- Software Verification and Testing with the ESBMC Framework
- Towards Self-Healing Software via Large Language Models and Formal Verification
- Towards Verification of Programs for CHERI Platforms with ESBMC

Deep Learning and Automated Program Repair



[1] Jin M, Shahriar S, Tufano M, Shi X, Lu S, Sundaresan N, Svyatkovskiy A. InferFix: End-to-End Program Repair with LLMs. arXiv e-prints. 2023 Mar:arXiv-2303.
[2] Li Y, Wang S, Nguyen TN. Dlfix: Context-based code transformation learning for automated program repair. InProceedings of the ACM/IEEE 42nd International Conference on Software Engineering 2020 Jun 27 (pp. 602-614).
[3] Gupta R, Pal S, Kanade A, Shevade S. Deepfix: Fixing common c language errors by deep learning. In Proceedings of the aaai conference on artificial intelligence 2017 Feb 12 (Vol. 31, No. 1).

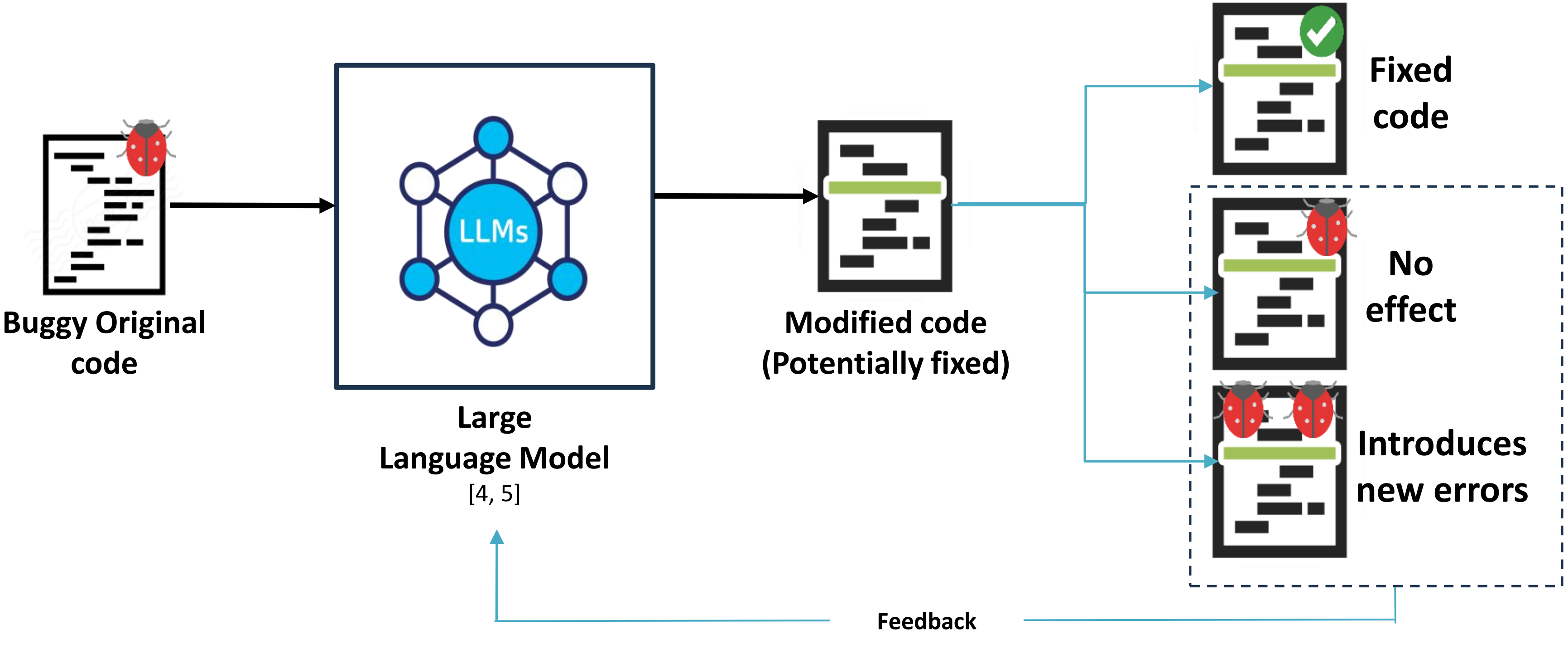
Large Language Models and Automated Program Repair

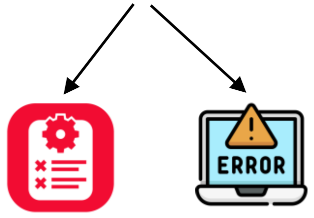


[4] Wang X, Wang Y, Wan Y, Mi F, Li Y, Zhou P, Liu J, Wu H, Jiang X, Liu Q. Compilable neural code generation with compiler feedback. arXiv preprint arXiv:2203.05132. 2022 Mar 10.

[5] Xia CS, Zhang L. Conversational automated program repair. arXiv preprint arXiv:2301.13246. 2023 Jan 30.

Large Language Models and Automated Program Repair

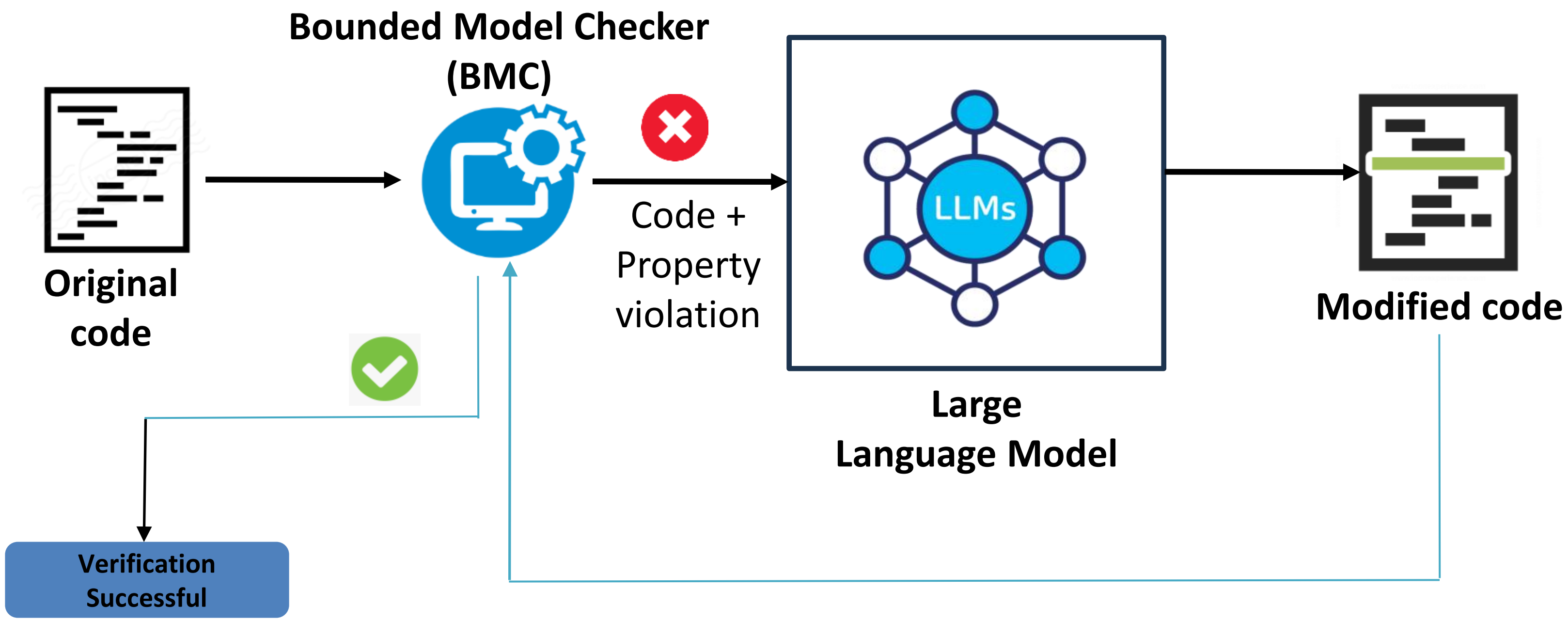




Compile-time error feedback misses run-time errors

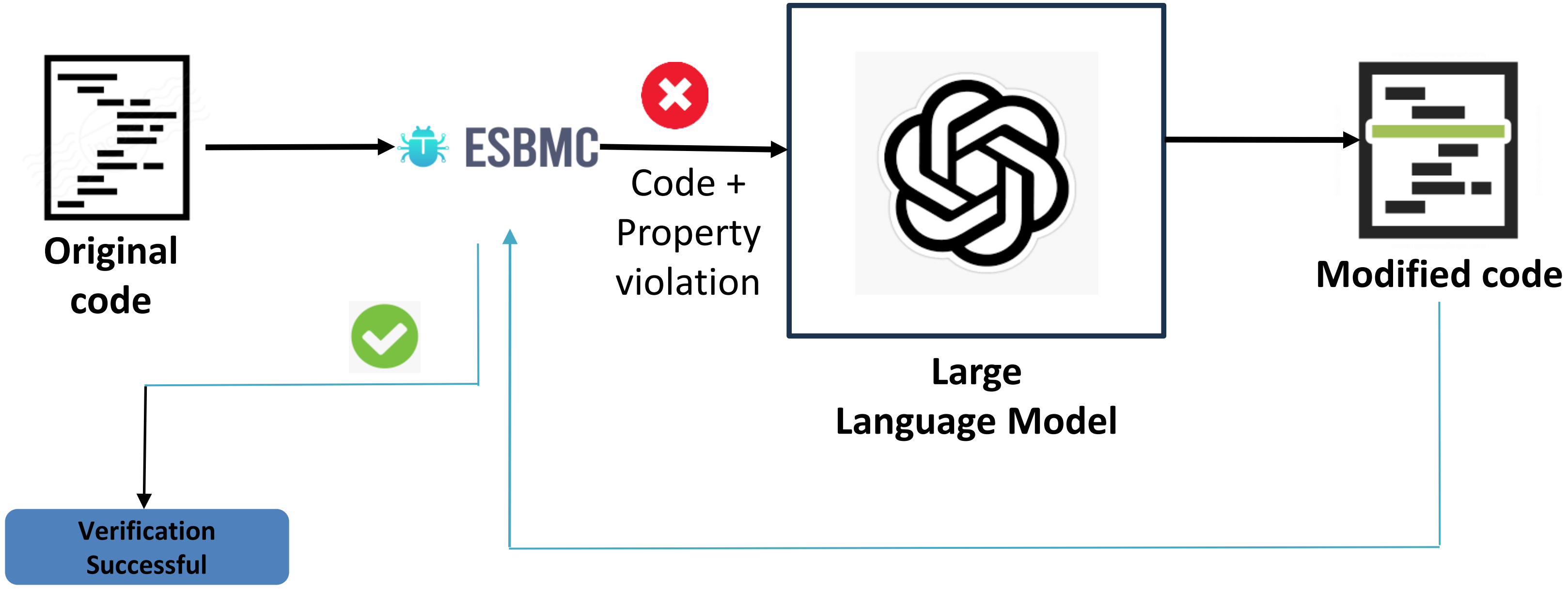
Test suite may not be available

LLM + Formal Verification for Self-Healing Software



[6] Charalambous, Y., Tihanyi, N., Jain, R., Sun, Y., Ferrag, M. Cordeiro, L.: A New Era in Software Security: Towards Self-Healing Software via Large Language Models and Formal Verification. Under review at the ACM Transactions on Software Engineering and Methodology, 2023.

LLM + Formal Verification for Self-Healing Software



LLM to Find Software Vulnerabilities

C++ program example

```cpp
int main() {
    int x=77;
    int y=x*x*x;
    int z=y*y;
    unsigned int r= z/1000;
    printf("Result %d\n", r);
    return 0;
}
```

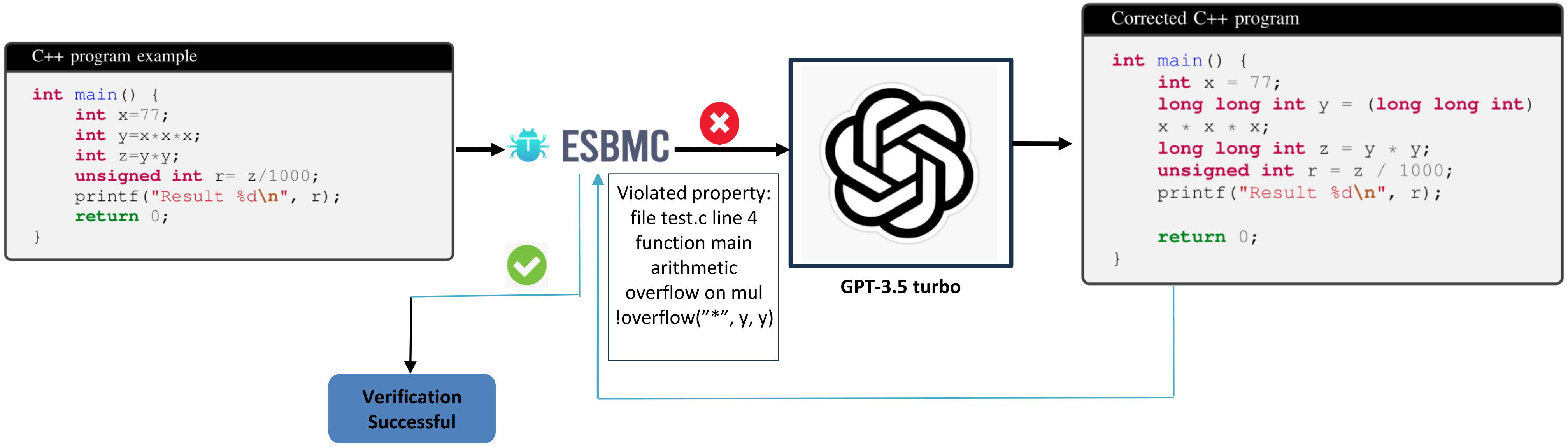
While we were in the process of preparing this presentation, if we asked GPT-3.5 *"Is there any problem with this code?"*, the response was an incorrect answer:

GPT-3.5 turbo

LLM answer:

"There is no problem with this code. The code correctly calculates the value of z as the square of y and then calculates the value of r as z divided by 1000. Finally, the code prints the result of r to the console. The calculations do not result in an integer overflow or any other issues, and the program should run correctly."

LLM + Formal Verification for Self-Healing Software



Experimental Evaluation

Set-up

Code Generation

- **Processor:** AMD Ryzen Threadripper PRO 3995WX
- **Cores:** 16
- **RAM:** 256 GB

Code Repair

- **Model:** MacBook Pro (2017)
- **RAM:** 16 GB RAM of LPDDR3 RAM (2133 MHz)
- **Processor:** 2.5 GHz Intel Core i7-7660U

Benchmarks

Generate 1000 programs with GPT-3.5 turbo with the following prompt

Code generation prompt

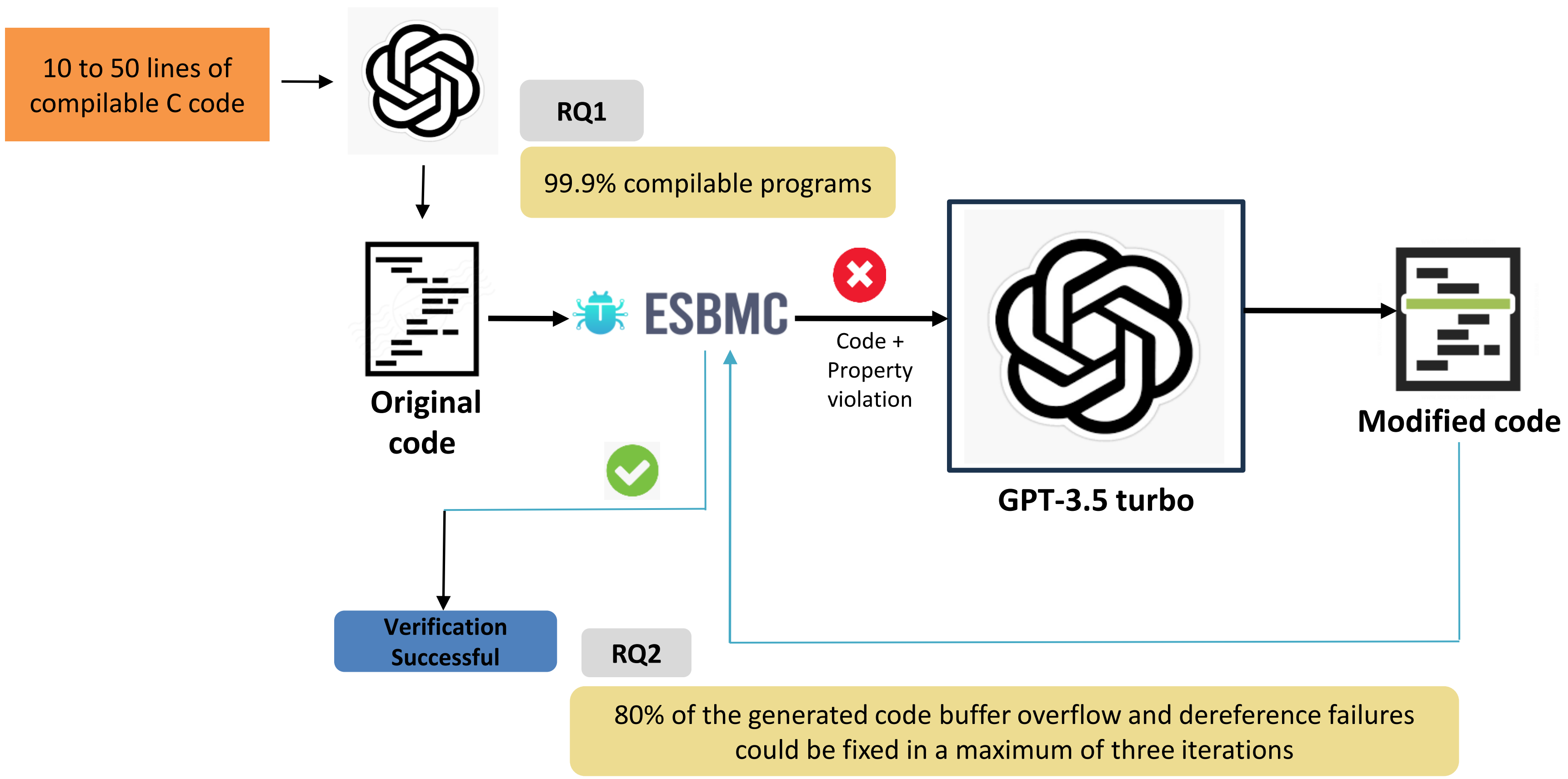
Generate a minimum of 10 and a maximum of 50 lines of C code. Use at least two functions. Use strings, arrays, bit manipulations, and string manipulations inside the code. Be creative! Always include every necessary header. Only give me the code without any explanation. No comment in the code.

Objectives

RQ1: (Code generation) Are the state-of-the-art GPT models capable of producing compilable, semantically correct programs?

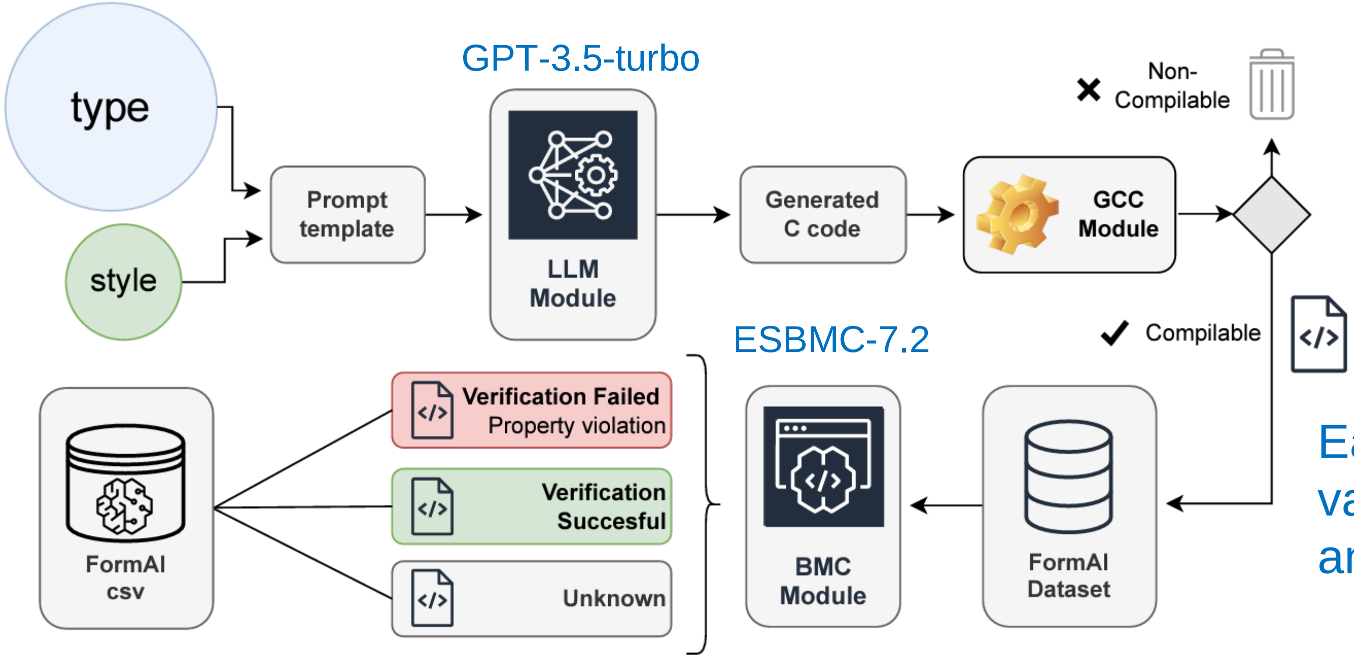
RQ2: (Code repair) Can external feedback improve the bug detection and patching ability of the GPT models?

Experimental Results



Generative AI through the Lens of Formal Verification

- The first AI-generated repository consisting of **112k independent and compilable C programs**

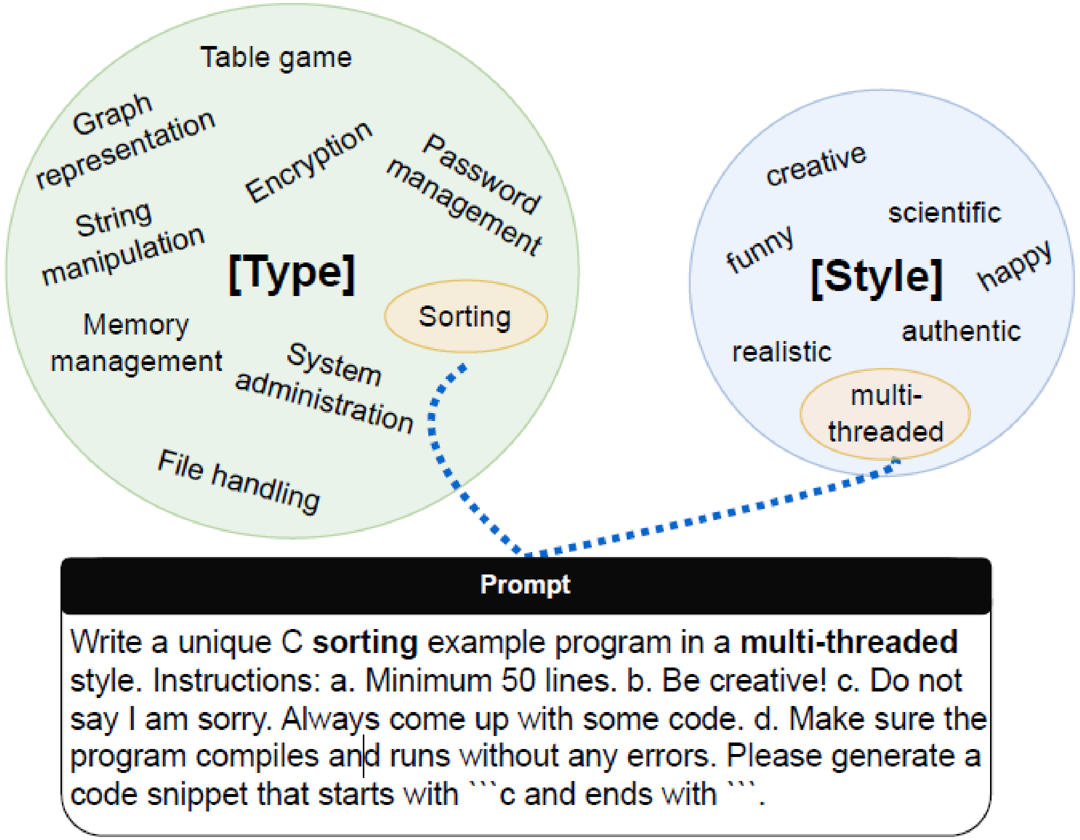


Each program varies between 50 and 600 lines

- Programming tasks from **network management and table games to string manipulation**

Tihanyi, N., Bisztray, T., Jain, R., Ferrag, M., Cordeiro, L., Mavroeidis, V.: The FormAI Dataset: Generative AI in Software Security Through the Lens of Formal Verification. Accepted at ACM PROMISE, 2023

Ensure Diversity



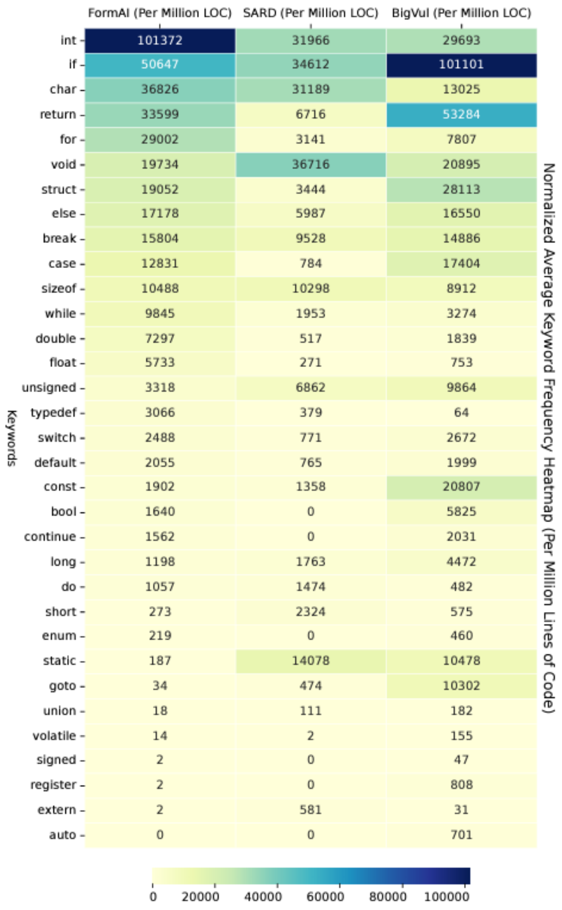
- Proper prompt engineering is crucial for achieving a diverse dataset
- Each API call randomly chooses a type from 200 options in the Type category, including topics like Wi-Fi Signal Strength Analyzer, QR Code Reader, and others
- Similarly, a coding style is selected from 100 options in the Style category during each query

Comparison of Various Datasets Based on their Labeling Classifications

Dataset	Only C	Source	#Code Snips.	#Vuln. Snips.	Multi. Vulns/Snip.	Comp./ Gran.	Vuln. Label.	Avg. LOC	Label. Method
Big-Vul	✘	Real-World	188,636	100%	✘	✘/Func.	CVE/CVW	30	PATCH
Draper	✘	Syn.+Real-World	1,274,366	5.62%	✔	✘/Func.	CWE	29	STAT
SARD	✘	Syn.+Real-World	100,883	100%	✘	✔/Prog.	CWE	114	BDV+STAT+MAN
Juliet	✘	Synthetic	106,075	100%	✘	✔/Prog.	CWE	125	BDV
Devign	✘	Real-World	27,544	46.05%	✘	✘/Func.	CVE	112	ML
REVEAL	✘	Real-World	22,734	9.85%	✘	✘/Func.	CVE	32	PATCH
DiverseVul	✘	Real-World	379,241	7.02%	✘	✘/Func.	CWE	44	PATCH
FormAI	✔	AI-gen.	112,000	51.24%	✔	✔/Prog.	CWE	79	ESBMC

Legend:
PATCH: GitHub Commits Patching a Vuln. **Man**: Manual Verification, **Stat**: Static Analyser, **ML**: Machine Learning Based, **BDV**: By design vulnerable

C Keyword Frequency and Associated CWEs



#Vulns	Vuln.	Associated CWE-numbers
88,049	$\mathcal{BOF}$	CWE-20, CWE-120, CWE-121, CWE-125, CWE-129, CWE-131, CWE-628, CWE-676, CWE-680, CWE-754, CWE-787
31,829	$\mathcal{DFN}$	CWE-391, CWE-476, CWE-690
24,702	$\mathcal{DFA}$	CWE-119, CWE-125, CWE-129, CWE-131, CWE-755, CWE-787
23,312	$\mathcal{ARO}$	CWE-190, CWE-191, CWE-754, CWE-680, CWE-681, CWE-682
11,088	$\mathcal{ABV}$	CWE-119, CWE-125, CWE-129, CWE-131, CWE-193, CWE-787, CWE-788
9823	$\mathcal{DFI}$	CWE-416, CWE-476, CWE-690, CWE-822, CWE-824, CWE-825
5810	$\mathcal{DFF}$	CWE-401, CWE-404, CWE-459
1620	$\mathcal{OTV}$	CWE-119, CWE-125, CWE-158, CWE-362, CWE-389, CWE-401, CWE-415, CWE-459, CWE-416, CWE-469, CWE-590, CWE-617, CWE-664, CWE-662, CWE-685, CWE-704, CWE-761, CWE-787, CWE-823, CWE-825, CWE-843
1567	$\mathcal{DBZ}$	CWE-369

- $\mathcal{ARO} \subseteq \mathcal{VF}$: Arithmetic overflow
- $\mathcal{BOF} \subseteq \mathcal{VF}$: Buffer overflow on `scanf()/fscanf()`
- $\mathcal{ABV} \subseteq \mathcal{VF}$: Array bounds violated
- $\mathcal{DFN} \subseteq \mathcal{VF}$: Dereference failure : NULL pointer
- $\mathcal{DFF} \subseteq \mathcal{VF}$: Dereference failure : forgotten memory
- $\mathcal{DFI} \subseteq \mathcal{VF}$: Dereference failure : invalid pointer
- $\mathcal{DFA} \subseteq \mathcal{VF}$: Dereference failure : array bounds violated
- $\mathcal{DBZ} \subseteq \mathcal{VF}$: Division by zero
- $\mathcal{OTV} \subseteq \mathcal{VF}$: Other vulnerabilities

The CWE Top 13

#	ID	Name
1	**CWE-787**	**Out-of-bounds Write**
2	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
4	**CWE-20**	**Improper Input Validation**
5	**CWE-125**	**Out-of-bounds Read**
6	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
7	**CWE-416**	**Use After Free**
8	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
9	CWE-352	Cross-Site Request Forgery (CSRF)
10	CWE-434	Unrestricted Upload of File with Dangerous Type
11	**CWE-476**	**NULL Pointer Dereference**
12	CWE-502	Deserialization of Untrusted Data
13	**CWE-190**	**Integer Overflow or Wraparound**

Which Parameters Are Most Effective?

Table: Classification results for different parameters

(u,t)	VULN	k-ind	Running time (m:s)	$\mathcal{VS}$	$\mathcal{VF}$	$\mathcal{TO}$	$\mathcal{ER}$
(2,1000)	2438	✗	758:09	371	547	34	48
(3,1000)	2373	✗	1388:39	366	527	57	50
(2,100)	2339	✗	175:38	367	529	61	43
(2,100)	2258	✓	400:54	340	603	20	37
(1,100)	2201	✗	56:29	416	531	17	36
(1,30)	2158	✓	146:13	349	581	34	36
(3,100)	2120	✗	284:22	354	483	120	43
(1,30)	**2116**	✗	**30:57**	**416**	**519**	**30**	**35**
(1,10)	2069	✓	61:58	360	553	52	35
(1,10)	2038	✗	19:32	413	503	51	33
(3,30)	1962	✗	125:19	342	444	172	42
(1,1)	1557	✓	10:59	355	406	208	31
(1,1)	1535	✗	6:22	395	374	201	30

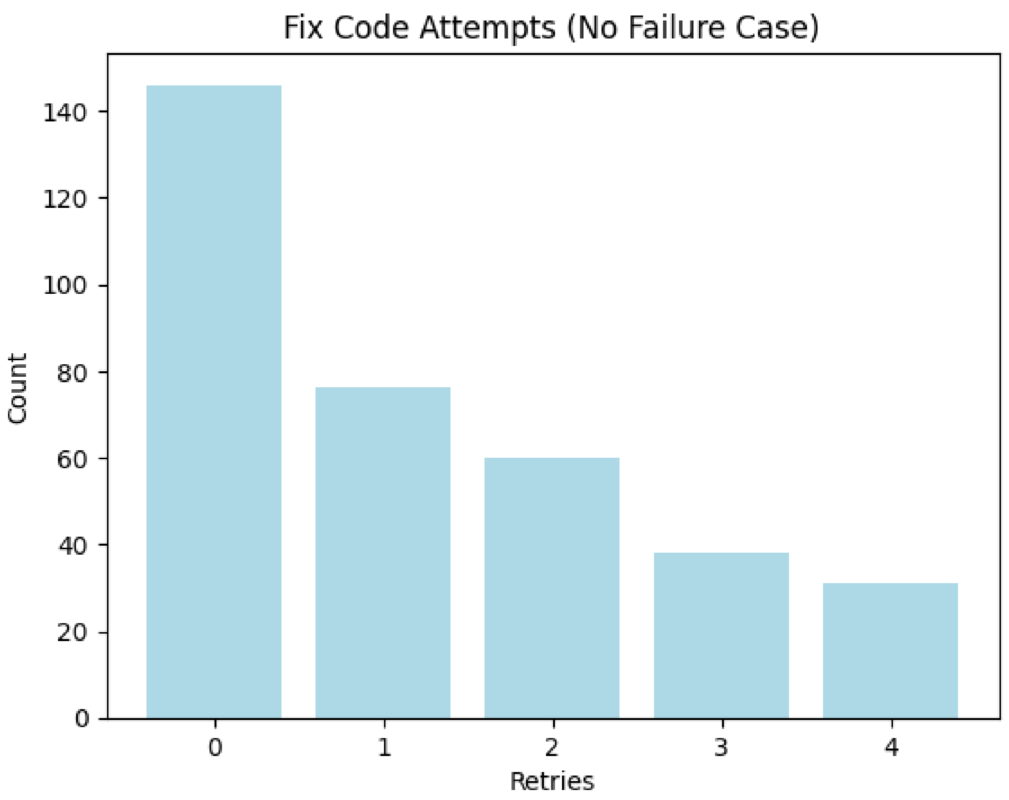
✓: Enabled, ✗: Disabled, (u, t) = unwind and timeout parameters

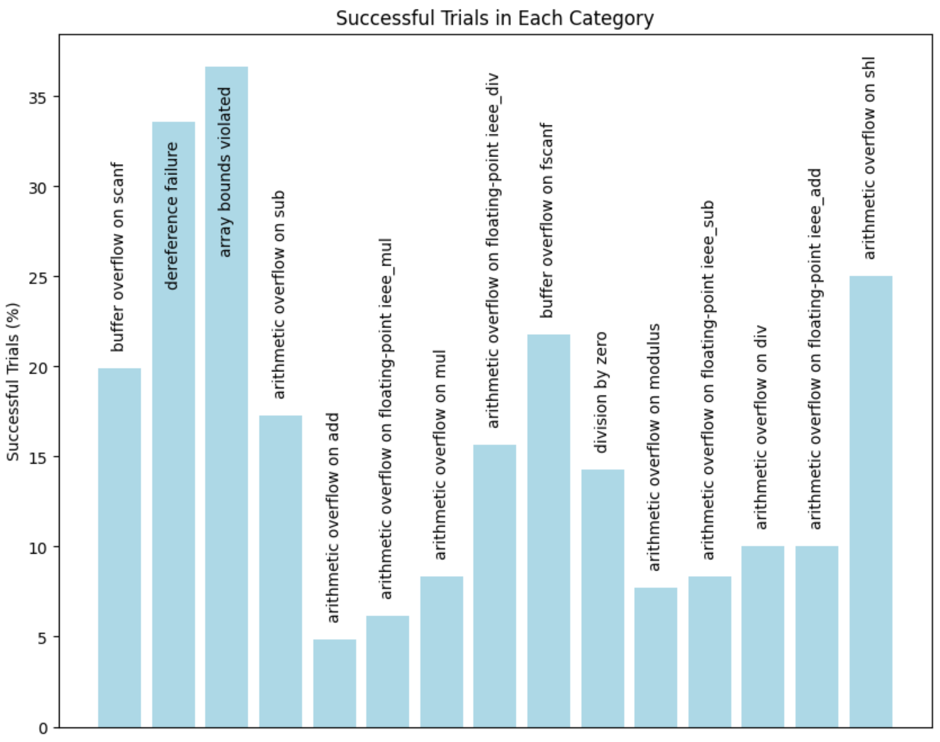
- We conducted experiments on 1,000 randomly selected samples
- The classification results showcase the effects of different unwind (u) and time (t) coupled with/without k-induction
- The detection results for parameter selection of *(u,t)=(1,10)*, *(1,30)*, or *(1,100)* without k-induction show that increasing the time threshold yields diminishing returns for the same unwind parameter

```
esbmc file.c --overflow --unwind 1 --memory-leak-check
--timeout 30 --multi-property --no-unwinding-assertions
```

Code Repair Performance

FormAI dataset	Accuracy
1000 samples randomly selected from 112k C programs	35.5%





FormAI Dataset - Availability

FORMAI DATASET: A LARGE COLLECTION OF AI-GENERATED C PROGRAMS AND THEIR VULNERABILITY CLASSIFICATIONS

Citation Author(s): Norbert Tihanyi (Technology Innovation Institute)
Tamas Bisztray (University of Oslo)
Ridhi Jain (Technology Innovation Institute)
Mohamed Amine Ferrag (Technology Innovation Institute)
Lucas C. Cordeiro (University of Manchester)
Vasileios Mavroeidis (University of Oslo)

Submitted by: Norbert Tihanyi
Last updated: Tue, 09/26/2023 - 05:10
DOI: 10.21227/vp9n-wv96
Data Format: *.csv (zip);

ACCESS DATASET | CITE | SHARE/EMBED

Categories: Artificial Intelligence, Security
Keywords: artificial intelligence, Software Vulnerability Dataset

WARNING: BE CAREFUL WHEN RUNNING THE COMPILED PROGRAMS, SOME CAN CONNECT TO THE WEB, SCAN YOUR LOCAL NETWORK, OR DELETE A RANDOM FILE FROM YOUR FILE SYSTEM. ALWAYS CHECK THE SOURCE CODE AND THE COMMENTS IN THE FILE BEFORE RUNNING IT!!!

https://github.com/FormAI-Dataset

DATASET FILES

- FormAI dataset: Vulnerability Classification (No C source code included) FormAI_dataset_human_readable-V1.csv (15.95 MB)
- FormAI dataset: 112000 compilable AI-generated C code FormAI_dataset_C_samples-V1.zip (97.61 MB)
- FormAI dataset: Vulnerability Classification (C source code included in CSV) FormAI_dataset_classification-V1.zip (60.66 MB)

Agenda

- Intoduce typical BMC Architectures for Verifying Software Systems
- Software Verification and Testing with the ESBMC Framework
- Towards Self-Healing Software via Large Language Models and Formal Verification
- Towards Verification of Programs for CHERI Platforms with ESBMC

Capability Hardware Enhanced RISC Instructions (CHERI)

63 … 0

permissions (15 bits)	reserved	base and bounds (41 bits)
pointer address (64 bits)		

CHERI 128-bit capability

CHERI Clang/LLVM and **LLD**[1] - compiler and linker for CHERI ISAs

[1]https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheri-llvm.html

CheriBSD[2] - adaptation of FreeBSD to support CHERI ISAs

[2]https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheribsd.html

ARM Morello[3] - SoC development board with a CHERI-extended ARMv8-A processor

[3]https://www.arm.com/architecture/cpu/morello

Mnemonic	Description
CGetBase	Move base to a GPR
CGetLen	Move length to a GPR
CGetTag	Move tag bit to a GPR
CGetPerm	Move permissions to a GPR
CGetPCC	Move the PCC and PC to GPRs
CIncBase	Increase base and decrease length
CSetLen	Set (reduce) length
CClearTag	Invalidate a capability register
CAndPerm	Restrict permissions
CToPtr	Generate **C0**-based integer pointer from a capability
CFromPtr	CIncBase with support for NULL casts
CBTU	Branch if capability tag is unset
CBTS	Branch if capability tag is set
CLC	Load capability register

CHERI-C program

```
#include <stdlib.h>
#include <string.h>
#include <cheri/cheric.h>

void main() {
   int n = nondet_uint() % 1024;                                /* models arbitrary user input */
   char a[n+1], *__capability b = cheri_ptr(a, n+1);
   b[n] = 17;                                                   /* succeeds */
   char *__capability c = cheri_setbounds(b-1, n);             /* fails: not the same object */
   /* ... */                                                    /* more CHERI-C API checks */
   memset_c(c, 42, n);                                          /* setting memory through a capability */
}
```

CHERI-C API

New capability types

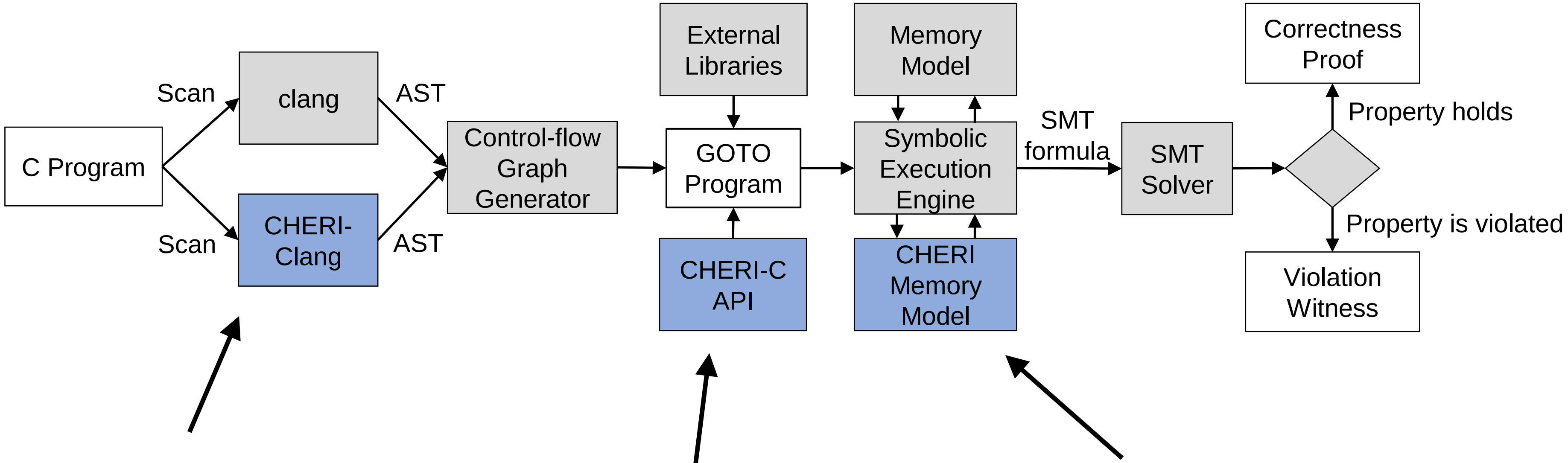
Pure-capability CHERI-C model

```c
#include <stdlib.h>
#include <string.h>
#include <cheri/cheric.h>

void main() {
   int n = nondet_uint() % 1024;
   char a[n+1], *__capability b = cheri_ptr(a, n+1);
   b[n] = 17;
   char *__capability c = cheri_setbounds(b-1, n);
   /* ... */
   memset_c(c, 42, n);
}
```

```c
#include <string.h>
#include <stdio.h>

void main(void) {
   int n = nondet_uint() % 1024;
   char a[n+1], *b = a;
   b[n] = 17;
   char *c = b-1;
   memset(c, 42, n);
}
```

All pointers are automatically replaced with capabilities by the CHERI Clang/LLVM compiler

ESBMC-CHERI



CHERI Clang/LLVM compiler

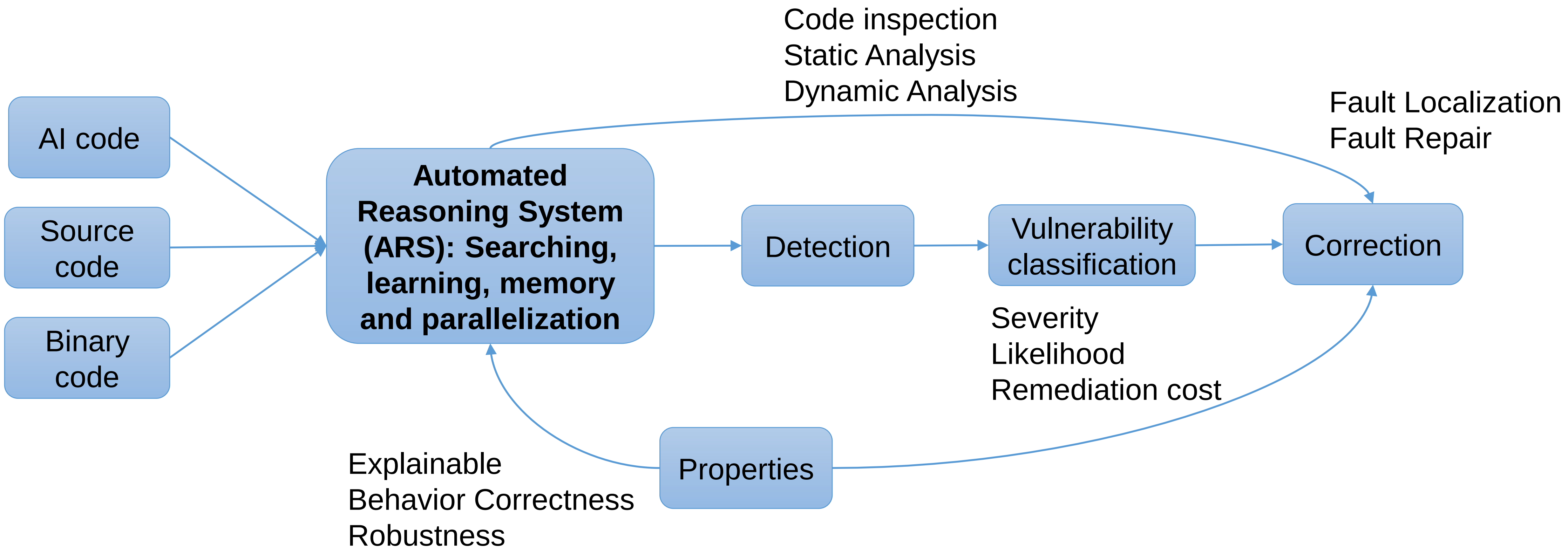
Implement computational model for CHERI-C API functions inside ESBMC (e.g., ***cheri_setbounds***)

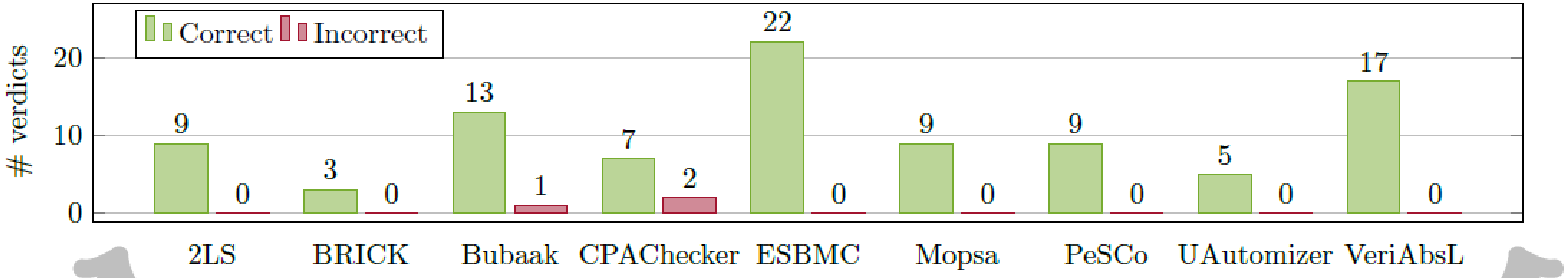
- New capability types
- Tagged memory
- Capability dereferencing

Braußе et al.: ESBMC-CHERI: towards verification of C programs for CHERI platforms with ESBMC. ISSTA 2022: 773-776

Vision: Automated Reasoning System for Secure SW and AI

Develop an automated reasoning system for safeguarding software and AI systems against security vulnerabilities in an increasingly digital and interconnected world





Preliminary results from SV-COMP'24 (may still change)

Existing software verifiers struggle on neural code!

Missing verdicts due to timeouts on many instances

Neural code challenges existing software verifiers:
float operations, calls to `math.h`, nested loops, multi-dimensional arrays.

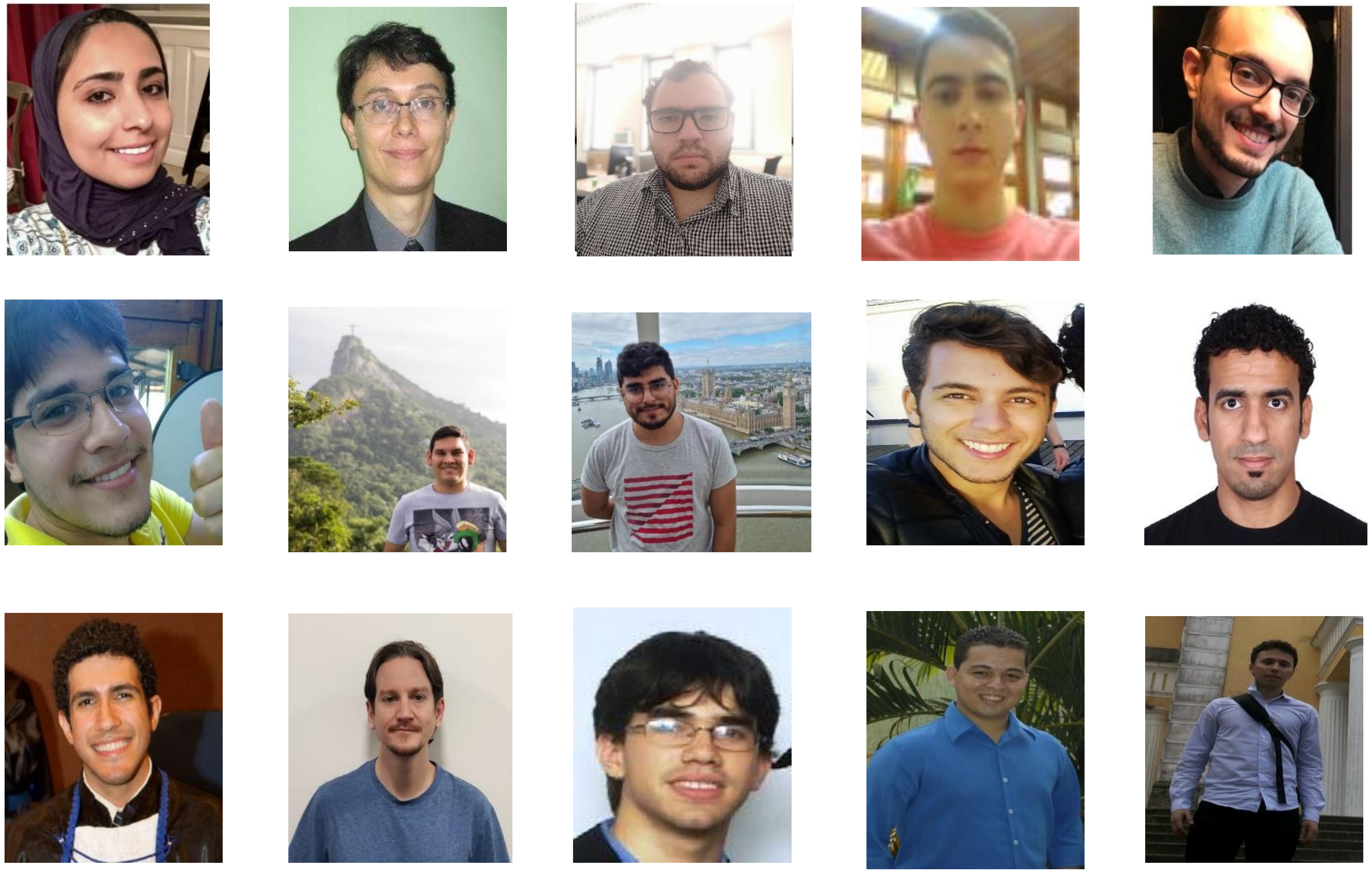
We release *NeuroCodeBench*, a benchmark of neural code verification:
6 categories, 14 functions from `math.h`, 32 neural networks, 607 properties;
safe/unsafe verdicts are either known a priori or independently verified.

Benchmark Category	Safe	Unsafe
`math_functions`	33	11
`activation_functions`	40	16
`hopfield_nets`	47	33
`poly_approx`	48	48
`reach_prob_density`	22	13
`reinforcement_learning`	103	193
Total	293	314

(Real) Impact: Students and Contributors

- 5 PhD theses
- 30+ MSc dissertations
- 30+ final-year projects

- GitHub:
  - 35 contributors
  - 22,160 commits
  - 212 stars
  - 84 forks



https://github.com/esbmc/esbmc

Impact: Awards and Industrial Deployment

- **Distinguished Paper Award** at ICSE'11
- **Best Paper Award** at SBESC'15
- **Most Influential Paper Award** at ASE'23
- **Best Tool Paper Award** at SBSeg'23
- **29 awards** from the international competitions on software verification (SV-COMP) and testing (Test-Comp) 2012-2023 at **TACAS/FASE**
  - Bug Finding and Code Coverage
- **Intel** deploys **ESBMC** in production as one of its verification engines for **verifying firmware in C**
- **Nokia and ARM** have found **security vulnerabilities** in **C/C++ software**
- **Funded by government** (EPSRC, British Council, Royal Society, CAPES, CNPq, FAPEAM) and **industry** (Intel, Motorola, Samsung, Nokia, ARM)

Acknowledgements

EPSRC
Engineering and Physical Sciences Research Council

UKRI
UK Research and Innovation

CNPq

CAPES

BRITISH COUNCIL

intel®

motorola

NOKIA

arm