MANCHESTER 1824
The University of Manchester

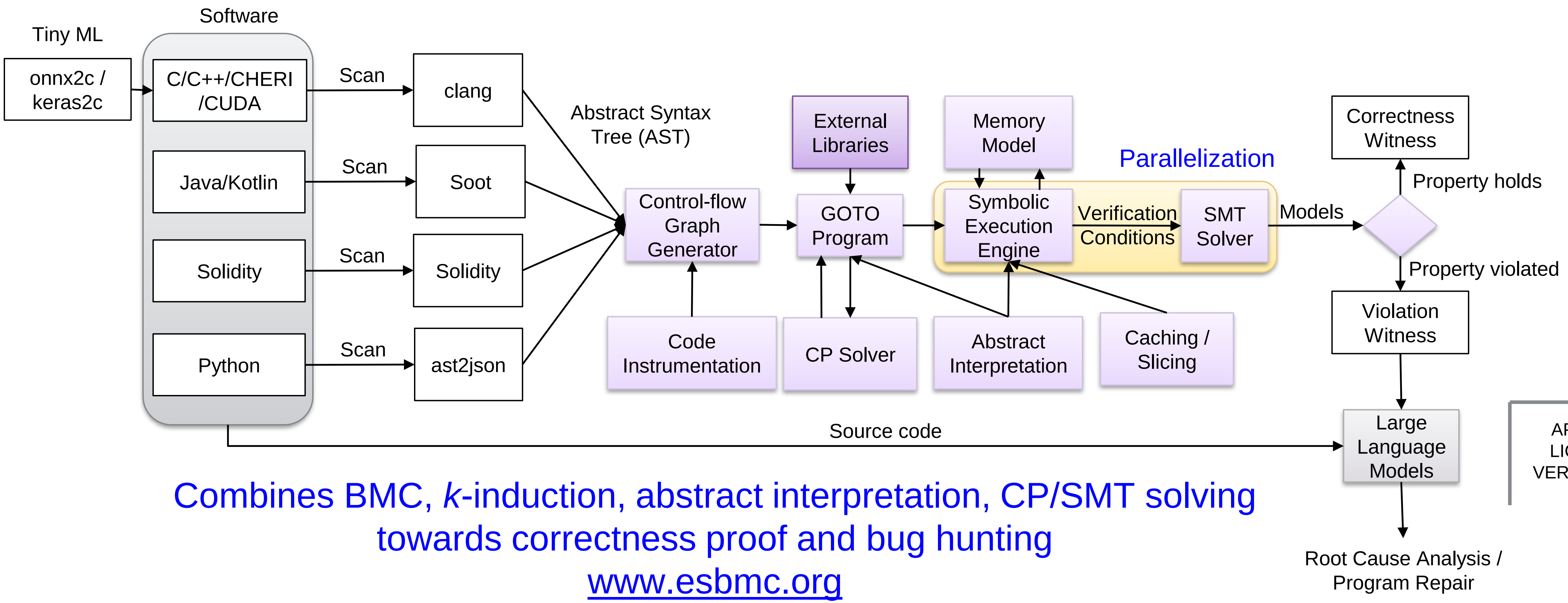
ESBMC v7.4: Harnessing the Power of Intervals

Rafael Menezes, Mohannad Aldughaim, Bruno Farias, Xianzhiyu Li, Edoardo Manino, Fedor Shmarov, Kunjian Song, Franz Brauße, Mikhail R. Gadelha, Norbert Tihanyi, Konstantin Korovin, and Lucas C. Cordeiro.

Project

- ESBMC is a verification engine capable of verifying C programs by relying on BMC, k-Induction and SMT.
- It is a joint project with the Federal University of Amazonas (Brazil), University of Southampton (UK), University of Manchester (UK), and University of Stellenbosch (South Africa).

ESBMC architecture



APACHE LICENSE VERSION 2.0

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

www.esbmc.org

Interval Analysis

Interval Analysis

- The interval analysis consists of computing all values the variables *might* assume at each statement.
- The analysis can be used to infer properties regarding the program states and flow.

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

Line	*Interval for "a"*
3	$(-\infty, +\infty)$
4	$(-\infty, 100]$
5	$(100, +\infty)$

Interval Analysis in BMC

- Interval analysis can help BMC by removing unreachable instructions:

```
int main() 0 ref
{
  int a; 2 refs
  if(a < 0 && a > 0)
    while(1) assert(0);
  return 0;
}
```

Contradiction

Unreachable

Interval Analysis in *k-induction*

- *k-induction* algorithm hijacks loop conditions to nondeterministic values, thus computing intervals become essential

```c
int main()
{
  unsigned int a = 10;
  unsigned int b = 1;

  while(a < 50 && *)
  {
    a++;
    b = a*2;
  }

  assert(b >= a);
}
```

```c
int main()
{
  unsigned int a = 10;
  unsigned int b = 1;

  a = *; b=*;
  while(a < 50 && *)
  {
    a++;
    b = a*2;
  }

  __ESBMC_assume(a < 50);
  assert((a*2) >= a);
}
```

Contracting Intervals

- The restrictions can be computed by using contractors (Forward/Backward)

```
int main()
{
  int a = * ? 1 : 11; // [1,11]
  int b = * ? 2 : 9; // [2,9]
  if(a < b) {
    // ...
  }
}
```

We can apply contractor algorithms to contract ”a” in terms of “b”:

Forward: $y = a - b \rightarrow [y] = ([a] - [b]) \cap (-infinity, 0]$

Backwards:

$$[a] = [a] \cap ([b] + [y])$$
$$[b] = [b] \cap ([a] - [y])$$

Forward: $[y]$ = [1,11]-[2,9] $\cap$ $(-infinity, 0]$ = = [-8,0]

Backwards:

$$[a] = [1{,}11] \cap ([2{,}9] + [-8{,}0]) = [1{,}9]$$
$$[b] = [2{,}9] \cap ([1{,}9] - [-8{,}0]) = [2{,}9]$$

ESBMC has support for other contractors by relying on ibex: a C++ numerical library based on **interval arithmetic** and **constraint programming.**

Computing Intervals

- For non-loop sequences:
  1. Initialize variable interval to $[(-\infty, \infty)]$;
  2. Use conditionals to restrict the interval;
  3. Merge intervals after conditionals;

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

Computing Intervals (contractor)

- Restrictions are computed through the use of contractors:
  - $[a] = [-infinity, 49]$
  - $[a] = [50, infinity]$
- 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

- For non-loop sequences:
  1. Initialize variable interval to $[(-\infty, \infty)]$.
  2. Use conditionals to restrict the interval.
  3. Merge intervals after conditionals.

Line	*Interval for "a"*
4	$(-\infty, +\infty)$
5	$(-\infty, 50)$
7	$[3,3]$
9	$[50, +\infty)$
11	$[5,5]$
12	$[3,5]$

```
1 int main()
2 {
3   int a;
4   if(a < 50) {
5     // ...
6     a = 3;
7   }
8   else {
9     // ...
10    a = 5;
11  }
12  // ...
13 }
```

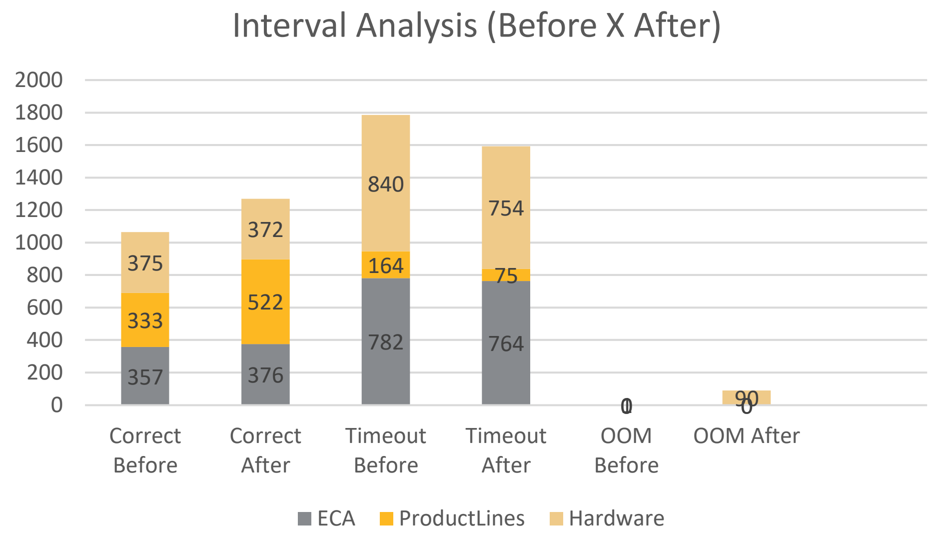
Computing Intervals

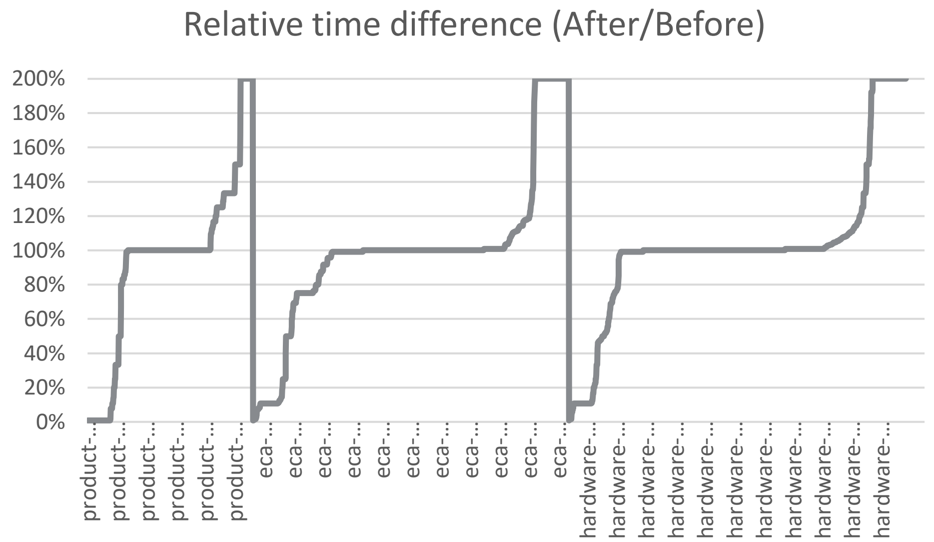
```c
int main()
{
  int a;
  if(a < 50) {
    // ...
    a = 3;
  }
  else {
    // ...
    a = 5;
  }
  // ...
}
```

```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);
  // ...
}
```

Results

- ✓ The instrumentation and optimizations helped the verification of unique tasks.
- × The preprocessing takes a toll in the hardware benchmarks.





Memory Leaks

Memtrack

- ESBMC employs a refined check for the valid-memtrack property.

1. At the end of an execution, for each memory object, add an assertion that it was deallocated correctly.
2. Add a guard into the assertion that there is no pointer currently referring to that memory object.

Memcleanup

- The new algorithm leverages the existing one tracking the lifetime of allocations for the valid-memcleanup property, but it specifically excludes still-reachable objects from the check.

Memory Object
Obj1
…
ObjN

Mem-cleanup checks if at the end of the execution, every memory object was freed.

Memcleanup

- The new algorithm leverages the existing one tracking the lifetime of allocations for the valid-memcleanup property, but it specifically excludes still-reachable objects from the check.

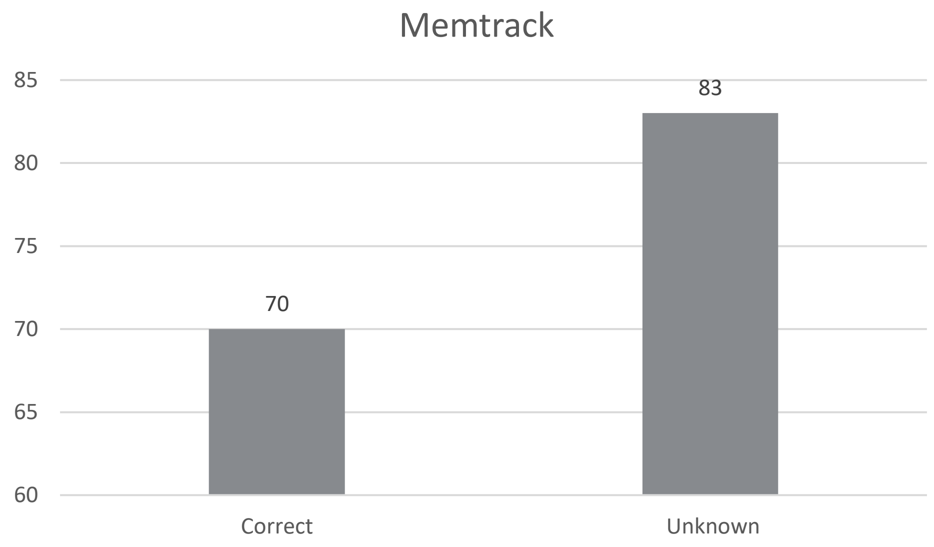
Memory Object
Obj1
…
ObjN

Pointer	Target
Ptr1	Obj1
…	…
PtrM	NULL

→ Obj1 check is removed!

Results

- The new algorithm to verify valid-memtrack benchmarks.
- There is a weakness in the current implementation concerning dynamic allocations only reachable through pointers stored in arrays of statically unknown size.
- We will address this weakness and submit suitable tasks for this property to SV-COMP in the future.



Math Operational Models

math.h

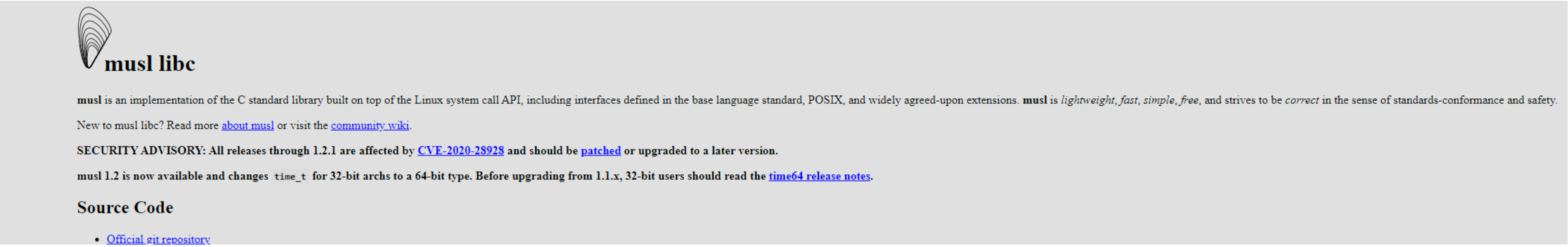
- ESBMC did not have precise OM for float operations. This used to be enough.
- The neural network benchmarks relies on 32-bit floats, which leaded to incorrect results.
- As a tradeoff between precision and verification speed, ESBMC now features a two-pronged design: precise and approximated.

math.h

- The IEEE 754 standard mandates bit-precise semantics for a small subset of the math.h library only (it includes: addition, multiplication, division, sqrt, fma, and other support functions such as remquo).
- In contrast, the behavior of most transcendental functions (e.g., sin, cos, exp, log) is platform-specific. Still, the standard recommends implementing the correct rounding whenever possible.

math.h

- For the most commonly-used float functions, we borrow the MUSL plain-C implementation of numerical algorithms.



math.h

- For the corresponding double functions, we employ less complex algorithms with approximate behavior.
  - For example, the exponential was approximated by Taylor series.

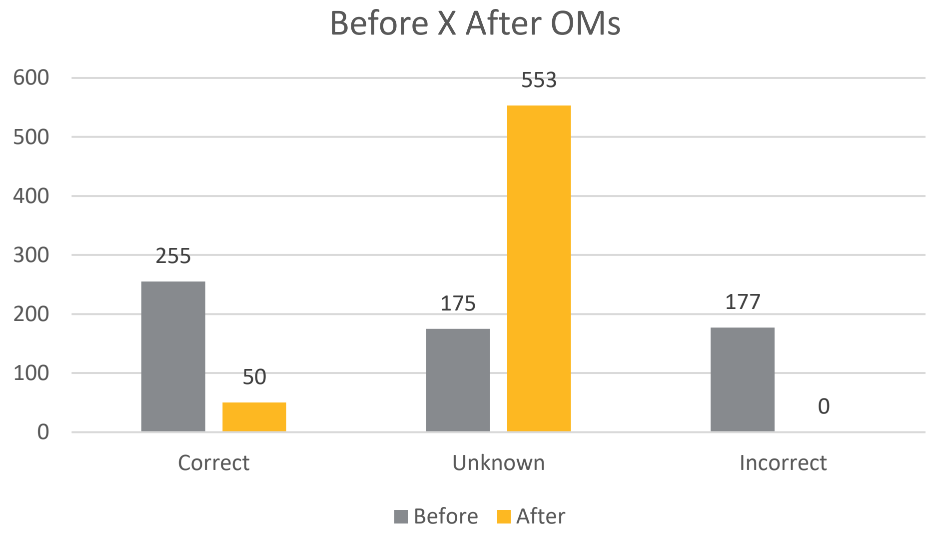
```c
double expm1(double x) /* exp(x) - 1 */
{
  switch(fpclassify(x))
  {
  case FP_NAN:
  case FP_ZERO:
    return x;
  case FP_INFINITE:
    return signbit(x) ? -1.0 : x;
  case FP_SUBNORMAL:
  case FP_NORMAL:
    break;
  }

  /* Taylor series converges everywhere, but the rate of convergence
   * is pretty bad; below we do a simple range reduction for larger |x|.
   */
  if(fabs(x) < 0x1p-3)
    return expm1_taylor(x);

  /* range reduction: exp(xm * 2^xe) = exp(xm) ^ (2^xe) */
  int xe;
  double xm = frexp(x, &xe); // |xm| in [2^-1,2^-0)
  xm *= 0x1p-3;
  xe += 3;                          // |xm| in [2^-4,2^-3)
  double r = expm1_taylor(xm) + 1; // r = exp(xm)
  /* xe is > 0 and xe < 1025+3, square xe times to account for 2^xe */
  for(int i = 0; i < xe; i++)
    r *= r;
  return r - 1;
}
```

Results

- Without operational models of the math.h library, ESBMC would assign non-deterministic results, which may cause incorrect counterexamples to be returned.
- Providing explicit operational models for many common functions in math.h improved the results



Data Races

Data Race Instrumentation

- ESBMC can detect data-races by instrumenting explicit assertions in the program.
- The instrumentation consists in setting an intermediate boolean for the assigned variable to true and then setting it back to false after the write is finished.
- The check consists in verifying whether there is a read while the write has not finished.
- The GOTO approach can increase the number of interleavings and has partial support for benchmarks that rely on pointer dereference.

Data Race Instrumentation

```
int foo() {
  global_var = 2;
}

int bar() {
  int tmp = global_var + 1;
  global_var = tmp;
}
```

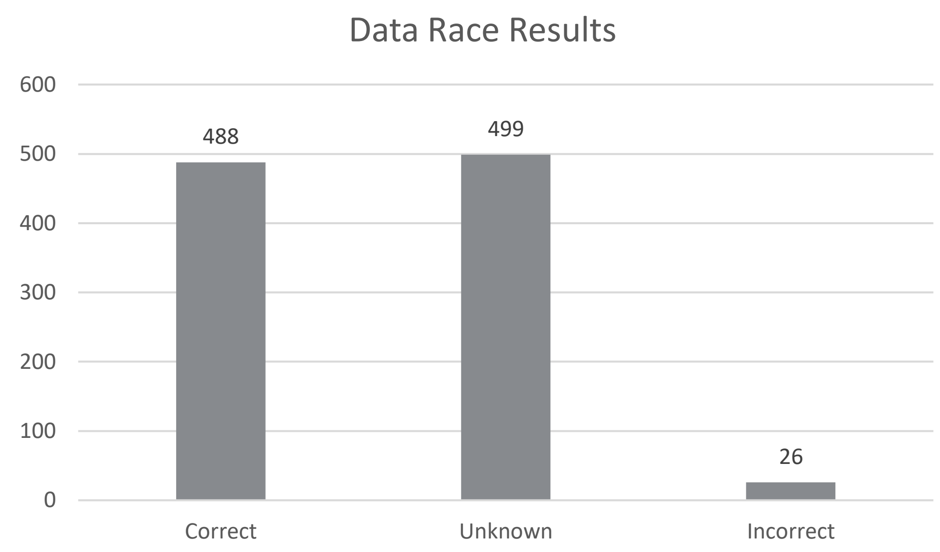
```
int foo() {
  global_var_writing = 1;
  global_var = 2;
  global_var_writing = 0;
}

int bar() {
  assert(!global_var_writing);
  int tmp = global_var + 1;
  global_var = tmp;
}
```

- Assuming that **foo** and **bar** are running in different threads, the assertion will check whether there is an interleaving where a read will happen before the assignment happens.

Symbolic Execution

- To improve the analysis, the property is now hybrid.
- The incorrect verdicts are mostly due to still missing support for detecting data races during dereferences of pointers to compound types.



Thanks for watching!