

# **Goblint & GobPie**

**Towards Usable Data Race Verification**

**Vesal Vojdani, CHES Industry Day**

**with Karoliine Holter, Simmo Saan (Uni Tartu)**

**and Julian Erhard, Sarah Tilscher, Michael Schwarz, Helmut Seidl (TU Munich)**

# Multiple Access data race

Two threads simultaneously access same memory location...

```
T1 : lock(&l1);  
      v = v + 1;  
      unlock(&l1);  
  
T2 : lock(&l1);  
      v = v + 1;  
      unlock(&l1);
```

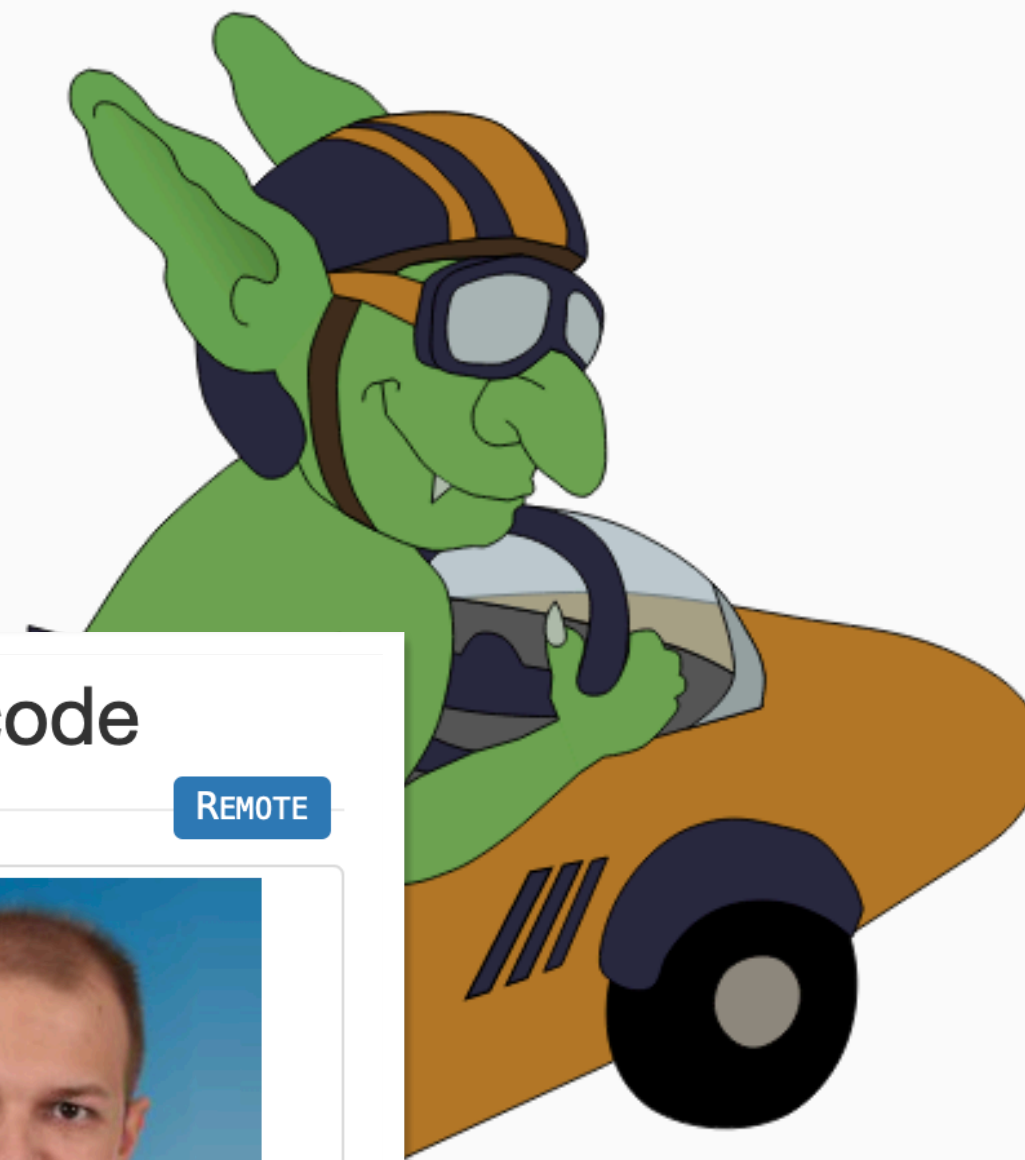
- List of accesses:  
  $\langle v, \{l_1\}, \text{write}, \text{file.c} : 2 \rangle$   
  $\langle v, \{l_1\}, \text{write}, \text{file.c} : 5 \rangle$
- $v$  is protected by  $\{l_1\}$ .

```
T1 : lock(&l1);  
      v = v + 1;  
      unlock(&l1);  
  
T2 : lock(&l2);  
      v = v + 1;  
      unlock(&l2);
```

- List of accesses:  
  $\langle v, \{l_1\}, \text{write}, \text{file.c} : 2 \rangle$   
  $\langle v, \{l_2\}, \text{write}, \text{file.c} : 5 \rangle$
- No common lock!

# Goblint: Race Detection World Champion

## Potentially somewhat useful as well ...



1. Goblint (1304)
2. Deagle (1211)
3. Dartagnan (768)
4. UAutomizer (756)
5. UGemCutter (732)
6. UTaipan (612)
7. CPAchecker (400)
8. Locksmith (226)
9. Theta (205)
10. ...

### Targeted Static Analysis for OCaml C Stubs: Eliminating gremlins from the code

REMOTE

Track [OCaml 2023](#)

When **Sat 9 Sep 2023 10:07 - 10:30** at [Grand Crescent](#) - [Session 1](#) Chair(s): [Benoît Montagu](#)

**Abstract** Migration to OCaml 5 requires updating a lot of C bindings due to the removal of naked pointer support. Writing OCaml user-defined primitives in C is a necessity, but is unsafe and error-prone. It does not benefit from either OCaml's or C's type checking, and existing C static analysers are not aware of the OCaml GC safety rules, and cannot infer them from existing macros alone. The alternative is automatically generating C stubs, which requires correctly managing value lifetimes. Having a static analyser for OCaml to C interfaces is useful outside the OCaml 5 porting effort too.

After some motivating examples of real bugs in C bindings a static analyser is presented that finds these known classes of bugs. The tool works on the OCaml abstract parse and typed trees, and generates a header file and a caller model. Together with a simplified model of the OCaml runtime this is used as input to a static analysis framework, Goblint. An analysis is developed that tracks dereferences of OCaml values, and together with the existing framework reports incorrect dereferences. An example is shown how to extend the analysis to cover more safety properties.

The tools and runtime models are generic and could be reused with other static analysis tools.



Edwin Török  
XenServer, Cloud Software Group

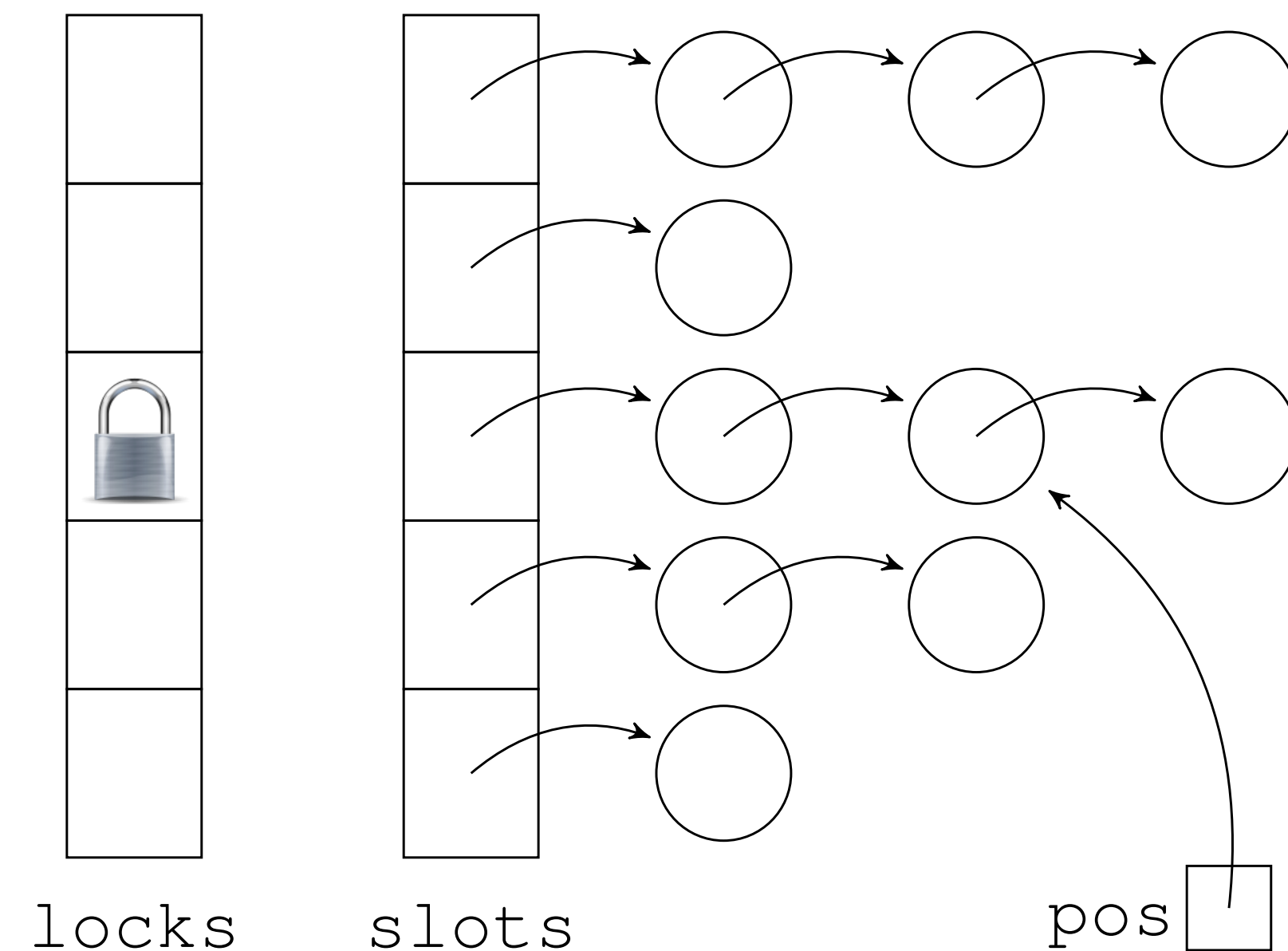
# Why just “somewhat useful”

- When running our SV-COMP configuration on top Github repos.
- Inconclusive results in 2h.
- One clearly needs precision adjustment...
- And targeted abstractions!

Goblint 2023-11-10 16:13:46 UTC goblint.svcomp.Concrat						
status	cputime (s)	walltime (s)	memory (MB)	safe	vulnerable	unsafe
Show all ▾	Min:Max	Min:Max	Min:Max	Min	Min:Max	Min:V
TIMEOUT	7200	7200	13800			
unknown	16.3	16.3	49.7	50	1	32
TIMEOUT	7200	7200	14200			
unknown	48.6	48.6	360	29	0	9
unknown	3480	3480	5730	118	0	6
TIMEOUT	7200	7200	9950			
TIMEOUT	7200	7200	979			
true	.186	.187	26.6	2	0	0
EXCEPTION (Stack overflow)	90.1	90.1	384			
EXCEPTION (Failure)	.173	.173	30.9			
unknown	2320	2320	11400	233	1	18
EXCEPTION (Stack overflow)	1120	1120	3160			
TIMEOUT	7200	7200	21600			
EXCEPTION (Stack overflow)	939	939	3240			
EXCEPTION (Stack overflow)	34.6	34.6	191			
EXCEPTION (Stack overflow)	1120	1120	7660			
SEGMENTATION FAULT	2320	2320	3880			
true	4.86	4.86	42.4	83	0	0

# Strength: Locks

- Infer the locked addresses  
 $\text{locks}[i]$
- Information about pointers  
 $\text{pos} \in \text{slot}[i]$
- Disjointness information  
 $\text{slot}[i] \cap \text{slot}[j] = \emptyset$



# Failings...

## Non-Locking Concurrency

- Thread-creation and joining
- Data segmentation
- When the number of threads is unbounded, this is hard...
- These “real-world” race challenges were submitted to SV-COMP!
- <https://github.com/goblint/bench/blob/master/concrat/race-challenges/README.md>

```
int *datas;

void *thread(void *arg) {
    int i = (int)arg;
    datas[i] = ...; // No locking needed
    return NULL;
}

int main() {
    int threads_total = __VERIFIER_nondet_int();

    pthread_t *tids = malloc(threads_total * sizeof(...));
    datas = malloc(threads_total * sizeof(int));

    // create threads
    for (int i = 0; i < threads_total; i++) {
        pthread_create(&tids[i], NULL, &thread, (void*)i);
    }

    // join threads
    for (int i = 0; i < threads_total; i++) {
        pthread_join(tids[i], NULL);
    }

    // compute with data – no locking needed
}
```

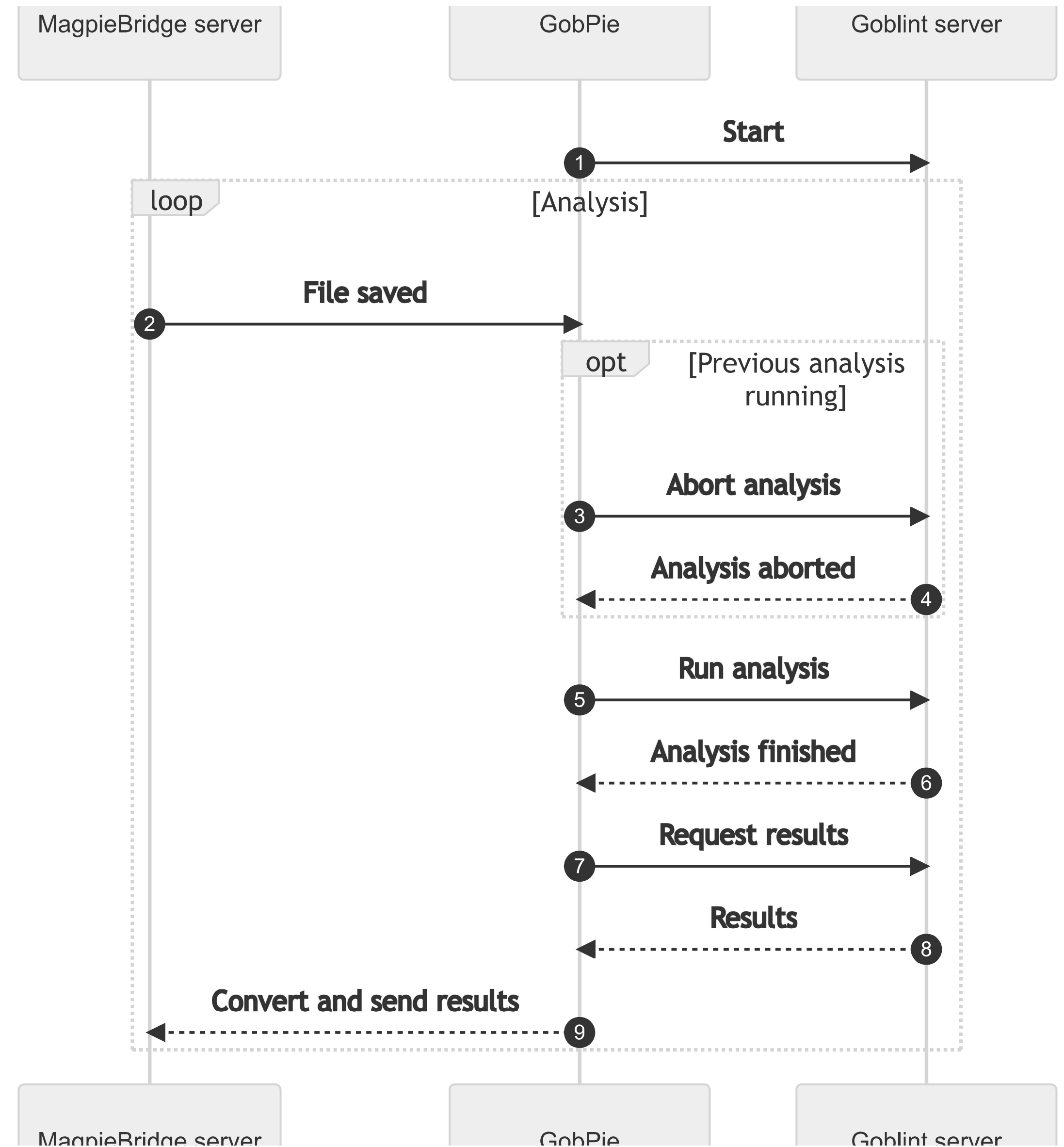
# Towards usable analysis

## Feedback from one industrial partner...

- False alarm can still be insightful and useful part of code review, but...
- “Why do you produce so many warnings about the same issue?”
- We (and perhaps others too) have not paid that much attention to explaining verification outcomes.
- Heuristic analyzers are much nicer and give actionable feedback.

# Interactive analysis

- Incremental abstract interpretation
- GUI integration via MagPie Bridge
- Server mode for Goblint
- Astronomical speedup (for superficial analysis)
- Modest speedup for SV-COMP quality analyses





# Abstract Debugging

More familiar experience of analysis results... (??)

The image shows the GobPie Abstract Debugger interface. On the left, the 'RUN AND DEBUG' panel displays the 'VARIABLES' section for a C program. It shows local variables 'a', 'b', 'c', and 'm' with their abstract values. 'a' is an unknown integer in the range [-31, 31] with a concrete value of 0. 'b' is an integer with a concrete value of 79. 'c' is a struct containing 'm' and 'n'. 'm' is an array of 3 integers, with the second element being an unknown integer in the range [79, 81]. 'n' is an unknown integer in the range [5, 131]. The 'WATCH' section shows expressions like 'c.m[2]', '(c.n \* 25 + 11) % 100', and 'c.n > 0' with their abstract values.

On the right, the source code for 'variables\_example.c' is shown. The code defines a struct 's' with an integer 'n' and an array of integers 'm'. The 'main' function initializes 'a' to a random value modulo 19, 'b' to 79, and 'c' to a struct with 'm' containing {7\*a+5, b, b+1, b+2}. The 'return 0;' line is highlighted in yellow.

```
RUN AND DEBUG C (GobPie Abstract Debugger)

VARIABLES
  Local
    a: [Unknown int([-31,31]), [0,18], Z]
    b: [79, [79,79], 79]
    c: {m: ..., n: [Unknown int([-31,31]), [5,131], 5+7Z]}
    m: {trivial arrays: [Not {0}([0,7]), [79,81], Z], length: [3, [...
      trivial arrays: [Not {0}([0,7]), [79,81], Z]
      length: [3, [3,3], 3]
    n: [Unknown int([-31,31]), [5,131], 5+7Z]
  Global
  Raw

WATCH
  c.m[2]: [Not {0}([0,7]), [79,81], Z]
  (c.n * 25 + 11) % 100: [Unknown int([-31,31]), [11,86], 11+25Z]
  c.n > 0: [1, [1,1], 1]

C variables_example.c > main()
2
3 struct s {
4     int n;
5     int m[3];
6 };
7
8 int main() {
9     int a = rand() % 19;
10    int b = 79;
11    struct s c = {
12        7 * a + 5,
13        {b, b + 1, b + 2}
14    };
15    return 0;
16 }
```