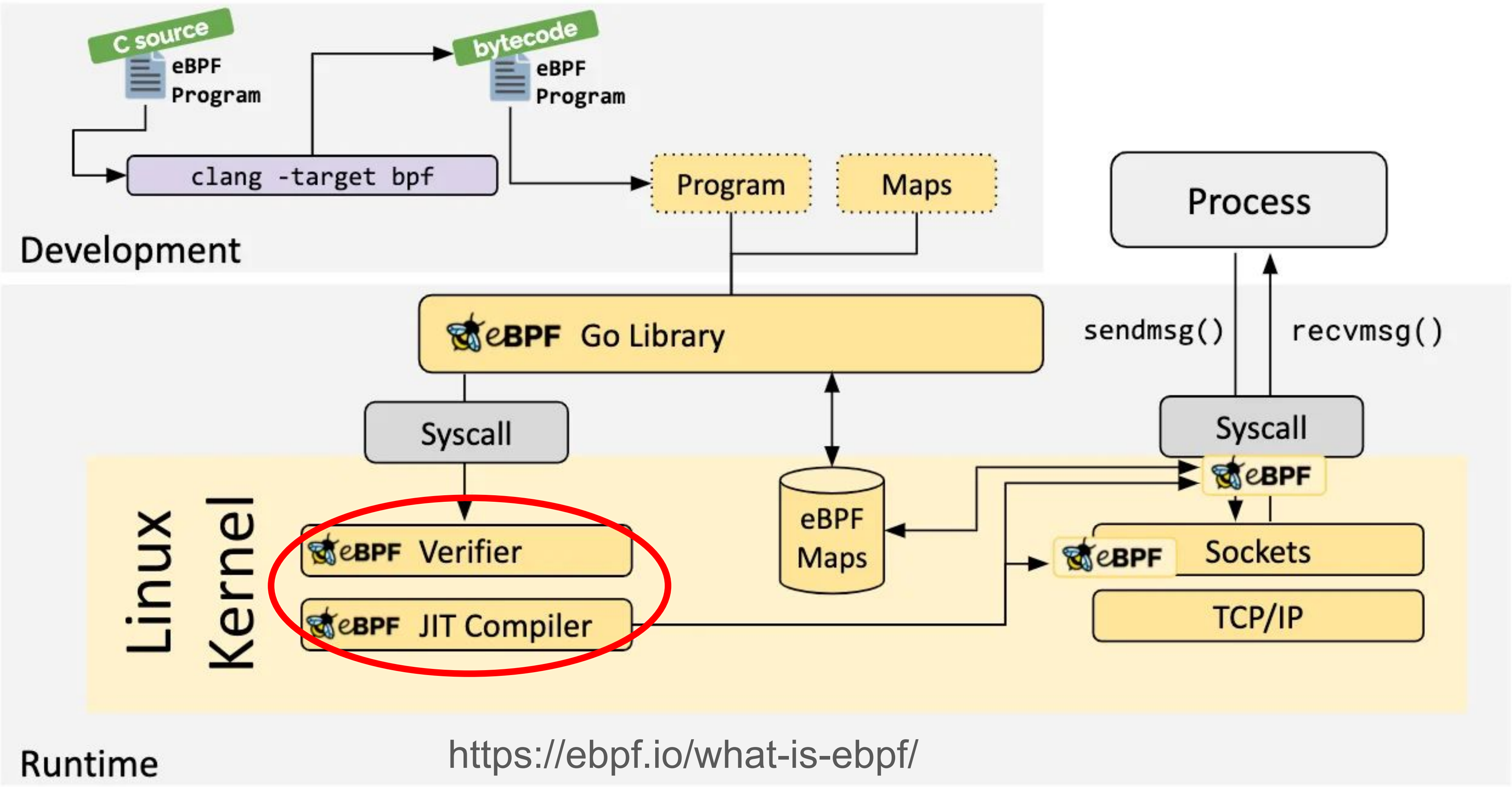
# eBPF Verifier 🔍

## Using declarative analysis to *try* to ensure safety

Shreyas Kallingal

# What is eBPF?

- Extended version of the Berkeley Packet Filter
- Allows sandboxed programs to run in an OS kernel 😨
- Observability, networking, security (really?)



https://ebpf.io/what-is-ebpf/

# eBPF Instruction Set

- Just a (very limited) 64-bit virtual machine!
- JiT compiler converts to native instructions
- Verifier *simply* needs to check safety before passing off to JiT

# eBPF Verifier “API”

- User provides eBPF bytecode to verifier
- Only context, stack space, and packets are available to VM
- Verifier is conservative (arguably, not enough)
- Loss of high-level source code information

# What is safety?

- Number one priority*
- No dead code
- Register readability (no reads before writes)
- **Pointer analysis**
- Termination

# Motivating Example

```
// r0 is a non-null pointer to a map value.
 // r1 initially can be any positive value on 64-bits.
 0: r6 = r0
 1: if r1 < 14 goto pc+1  // Jump to insn 3 if r1 is bounded.
 2: r1 &= 0xf             // If it is not, bound it.
 3: r6 += r1
 4: r7 = *(u16 *)(r6 + 0) // Read map value.
```

Source: https://pchaigno.github.io/ebpf/2023/09/06/prevail-understanding-the-windows-ebpf-verifier.html

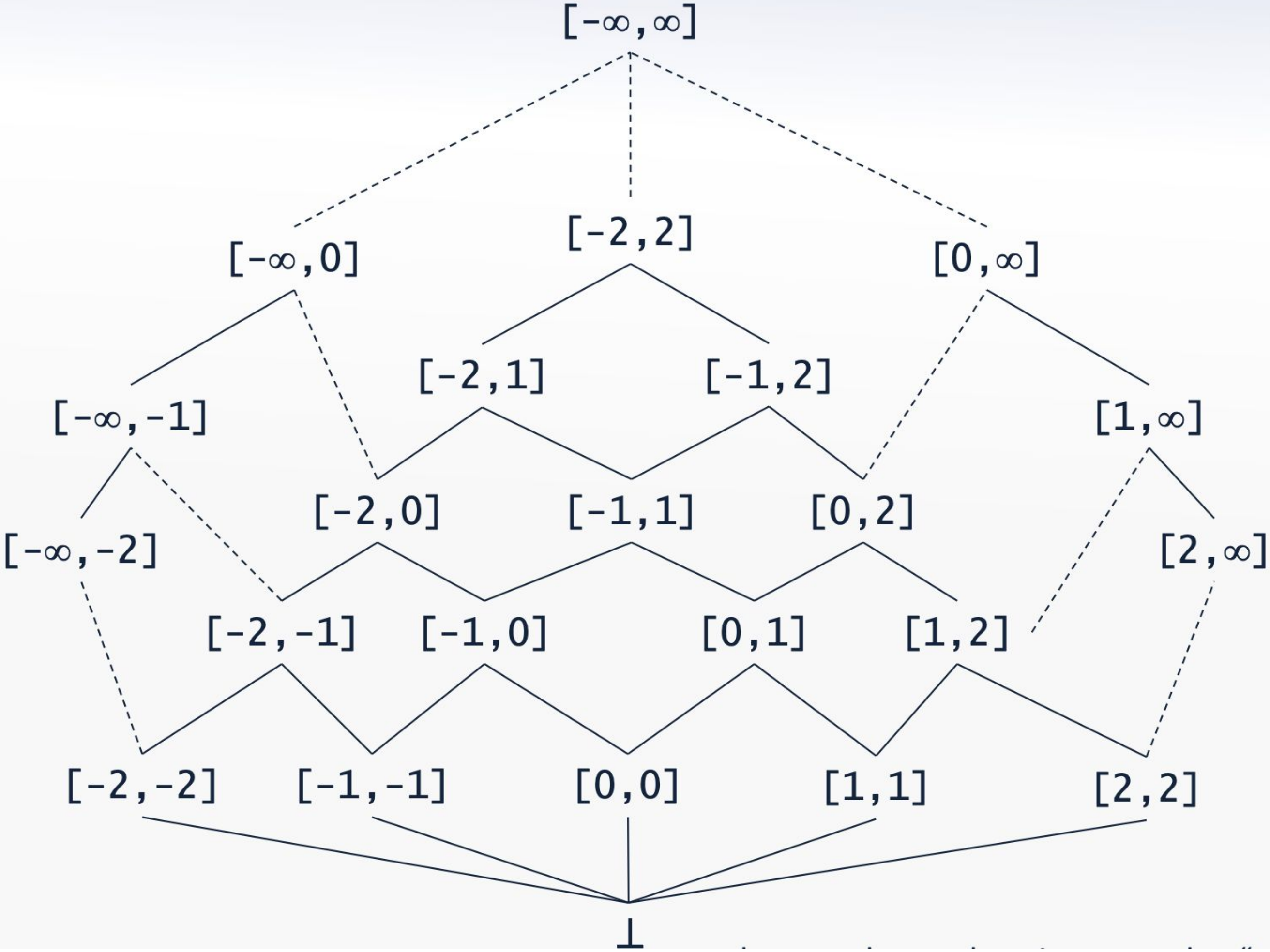
# Abstract Interpretation for Pointer Analysis

Fixed-point problem!

1. Start with a basic block input state from predecessors
2. Perform abstract interpretation over that block → new output state
3. Update successors
4. Rinse and repeat until you settle on a fixpoint

# Crab 🦀

- Fixed point solver
- Widening as a method for coarsening the interval analysis (overshoots)
- Specialized “CrabIR” used for its control flow analysis

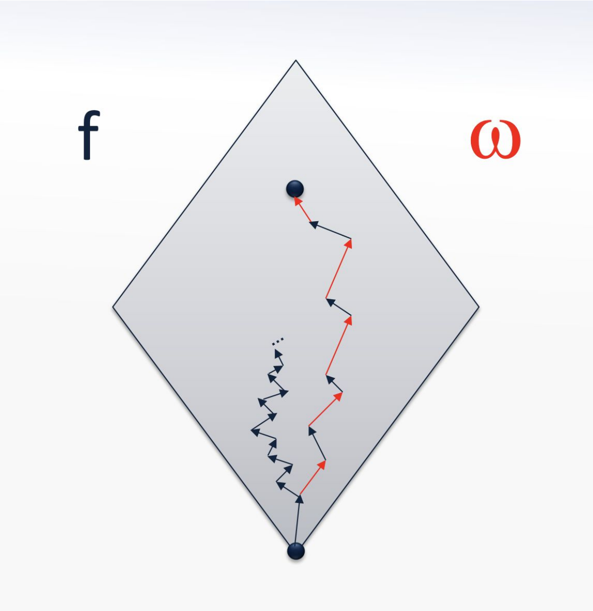

[-∞,∞]
[-∞,0]
[-2,2]
[0,∞]
[-2,1]
[-1,2]
[-∞,-1]
[1,∞]
[-2,0]
[-1,1]
[0,2]
[-∞,-2]
[2,∞]
[-2,-1]
[-1,0]
[0,1]
[1,2]
[-2,-2]
[-1,-1]
[0,0]
[1,1]
[2,2]
⊥

# Widening: A key optimization

For intervals:

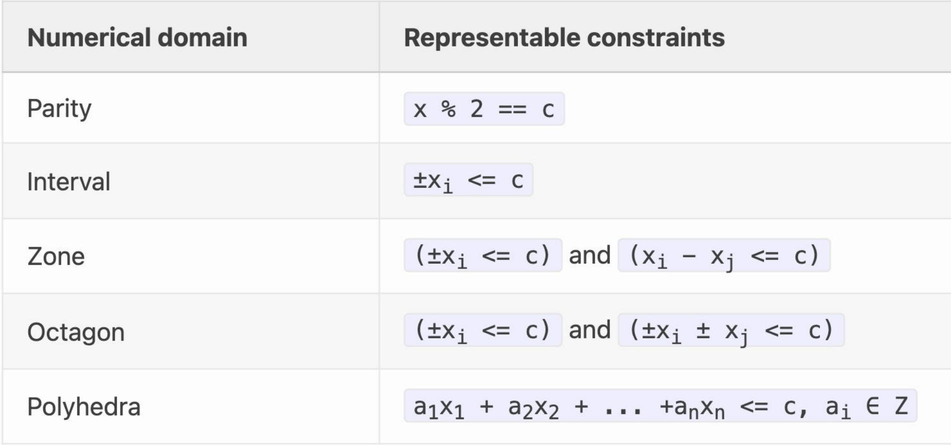
$$\omega([a,b]) = [\ \mathit{max}\{i \in B \mid i \leq a\},\ \mathit{min}\{i \in B \mid b \leq i\}\ ]$$

$$\omega(\bot) = \bot$$



# PREVAIL (2019)

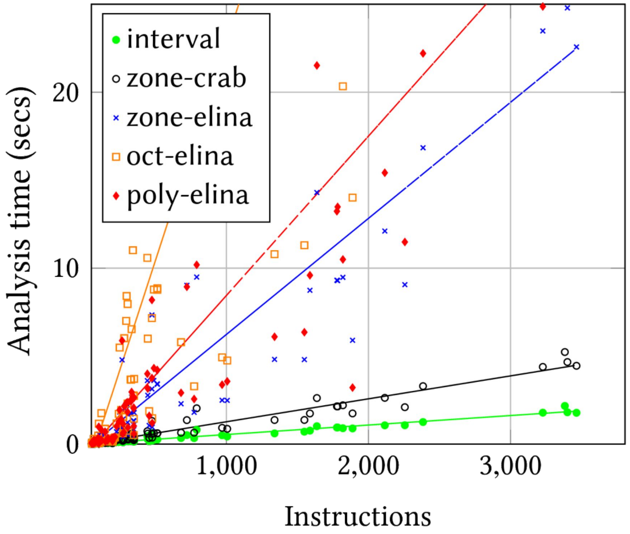
- De-facto Windows verifier built on Crab
- Leverages abstract interpretation to scale analysis to larger programs
- Domain must be relational to fully encompass run-time bounds checks

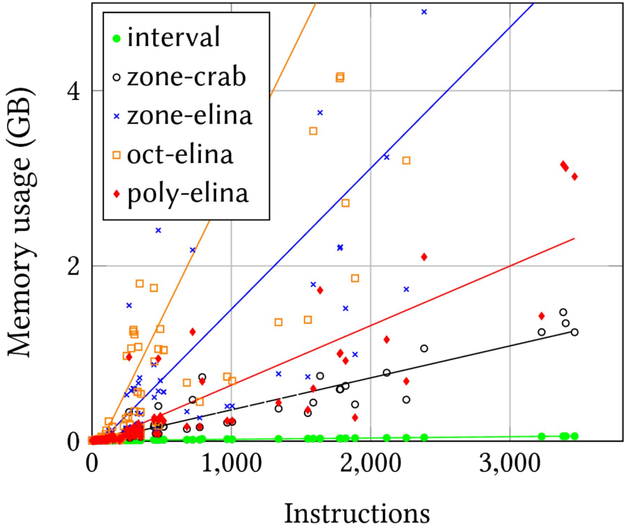


| Numerical domain | Representable constraints |
|---|---|
| Parity | `x % 2 == c` |
| Interval | $\pm x_i <= c$ |
| Zone | $(\pm x_i <= c)$ and $(x_i - x_j <= c)$ |
| Octagon | $(\pm x_i <= c)$ and $(\pm x_i \pm x_j <= c)$ |
| Polyhedra | $a_1x_1 + a_2x_2 + ... +a_nx_n <= c, a_i \in Z$ |

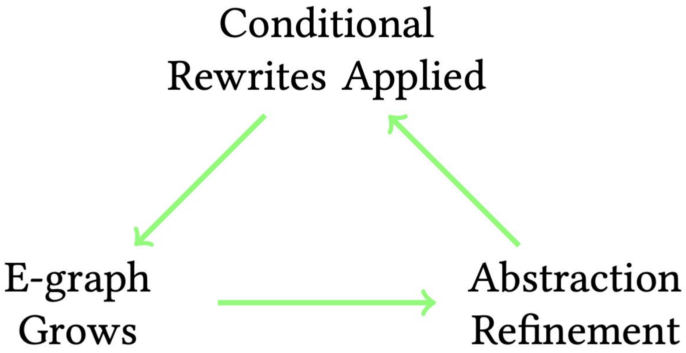
# Performance considerations

- PREVAIL is a resource hog; ~5X overhead over standard Linux verifier





# AI and eGraphs: Better Together

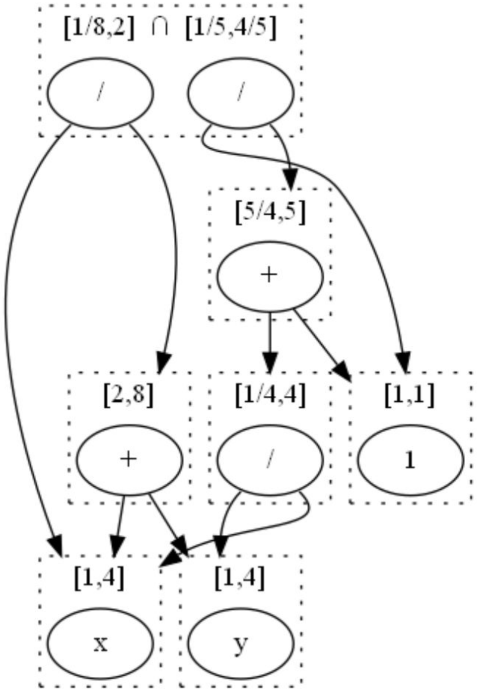


Conditional Rewrites Applied

E-graph Grows

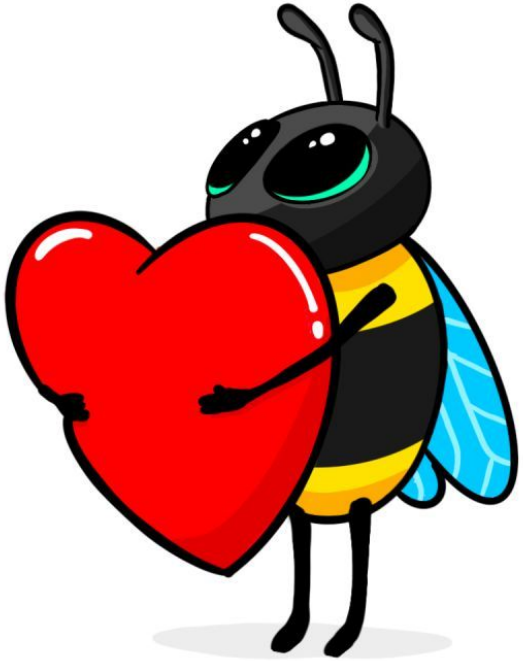
Abstraction Refinement

S. Coward, G. A. Constantinides, and T. Drane, “Combining E-Graphs with Abstract Interpretation.” arXiv, Aug. 15, 2023. doi: 10.48550/arXiv.2205.14989.



# Verifier Future (or Demise?)

- Formal verification of the verifier does exist
- Comparison to Wasm security models
- Argument that verification is untenable (Rust alternative)
- Runtime checking is a necessary evil



Questions?