Verifying the Verifier: eBPF Range Analysis Verification

Harishankar Vishwanathan
Matan Shachnai
Srinivas Narayana
Santosh Nagarakatte

RUTGERS

One Page Summary

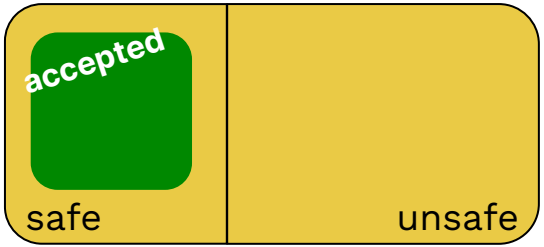
- The eBPF Verifier is crucial.
- Static Analysis is a crucial part of the eBPF verifier.
  - Writing correct static analysis is hard. Formal verification can be useful!
- Contributions: A tool, Agni:
  - Automatically checks the correctness of (part of) the static analysis in the eBPF verifier, on every commit.
- Results
  - Analyses in kernels starting from `5.13` — `5.19` are correct.
  - `4.14` — `5.12`: Agni reports bugs.
    - Agni can generate proof-of-concept eBPF programs that manifest bugs.
- Please give your feedback

RUTGERS

eBPF Verification Must be Sound

- **Soundness**: Unsafe programs should be rejected
- **Safety**:
  - Termination
  - Illegal operations
  - Memory access

Can we formally verify the soundness of the static analysis in the eBPF verifier?

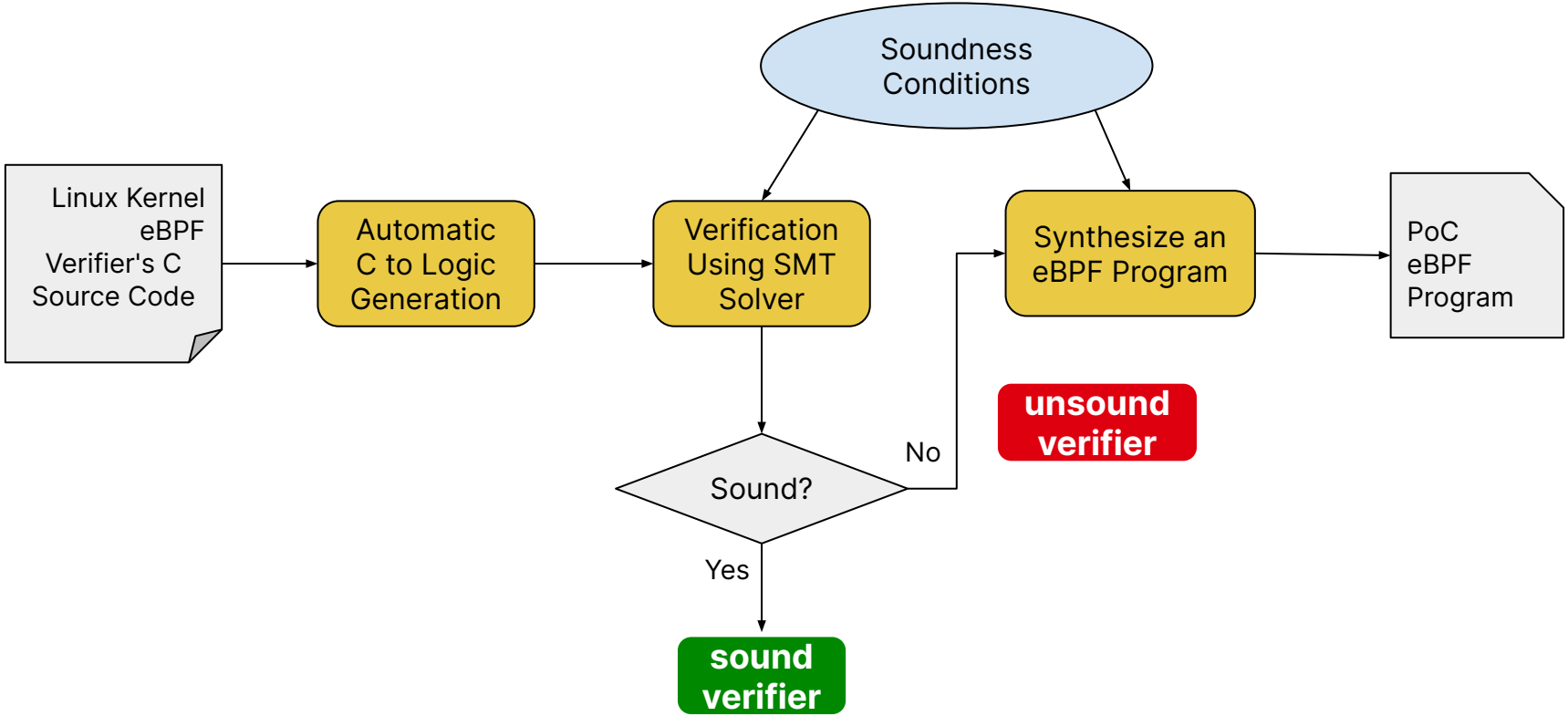


RUTGERS

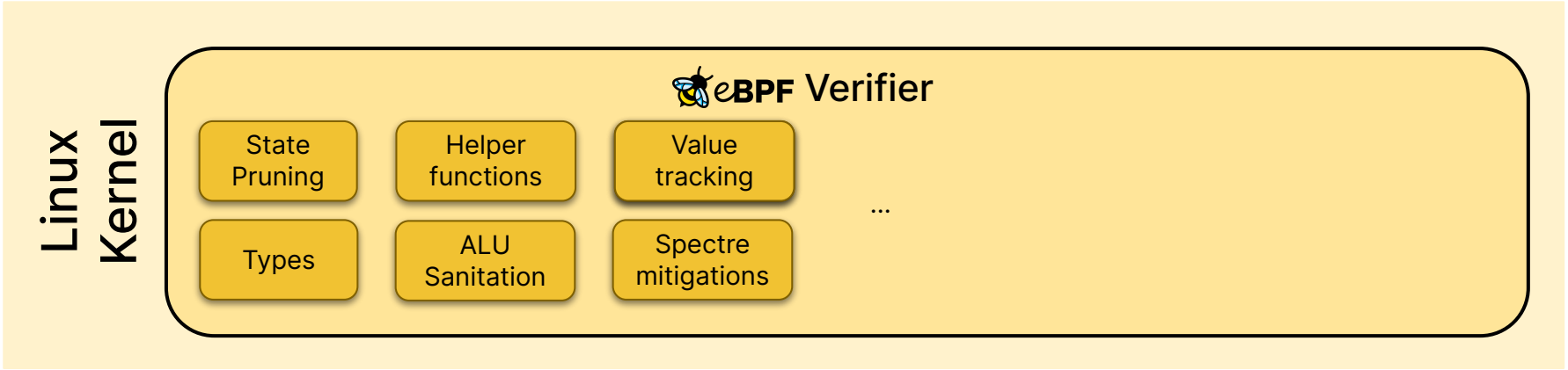
Goals

- Goal: Verify the soundness on every commit
  - Verifier is a large code base
  - Constantly changing
- Options:
  - Manually verify
    - Manually write the kernel code in Logic (SMT)
    - Repeat on every commit
    - Tedious and error prone
  - 🤖 Automate 🤖

Overview

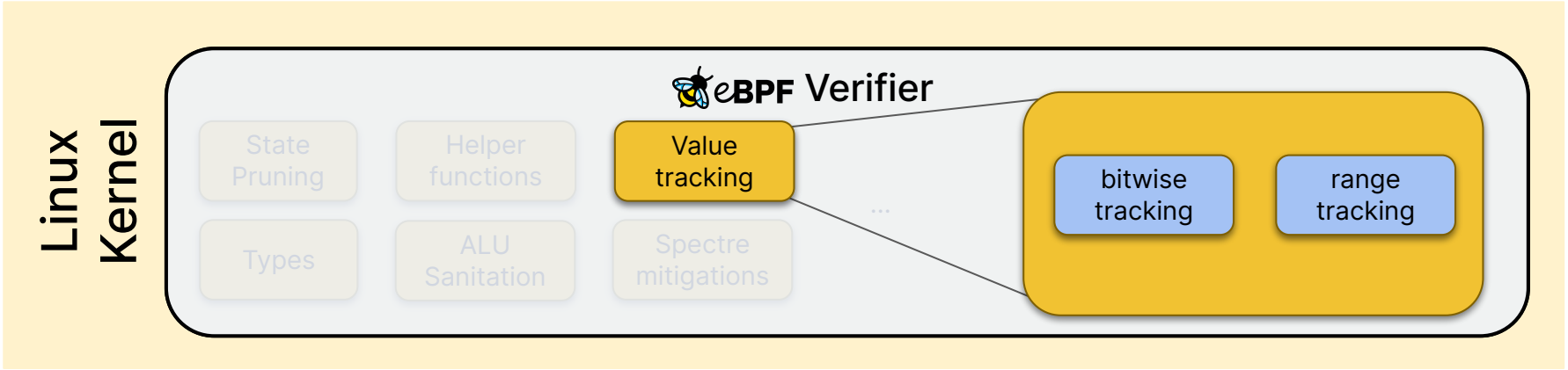

Soundness Conditions
Linux Kernel eBPF Verifier's C Source Code
Automatic C to Logic Generation
Verification Using SMT Solver
Synthesize an eBPF Program
PoC eBPF Program
Sound?
No
Yes
unsound verifier
sound verifier

Static Analyses in the eBPF verifier



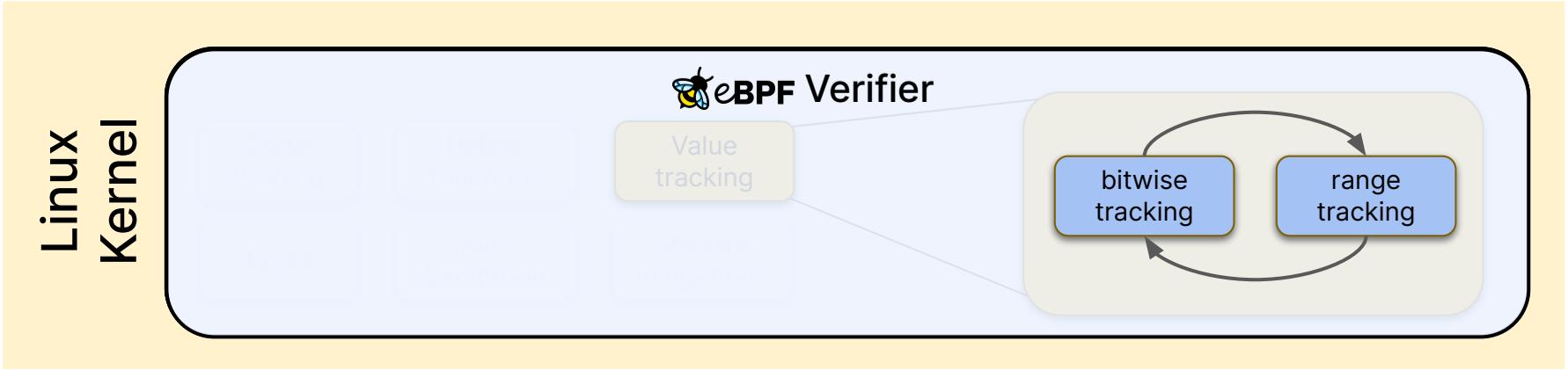
- Tracking the **values** of program variables across **all** executions of the program
- Our work: Reasoning about the soundness and precision of the range analysis + bitwise tracking + their combination

Static Analyses in the eBPF verifier



- Tracking the **values** of program variables across **all** executions of the program
- Our work: Reasoning about the soundness and precision of the range analysis + bitwise tracking + their combination

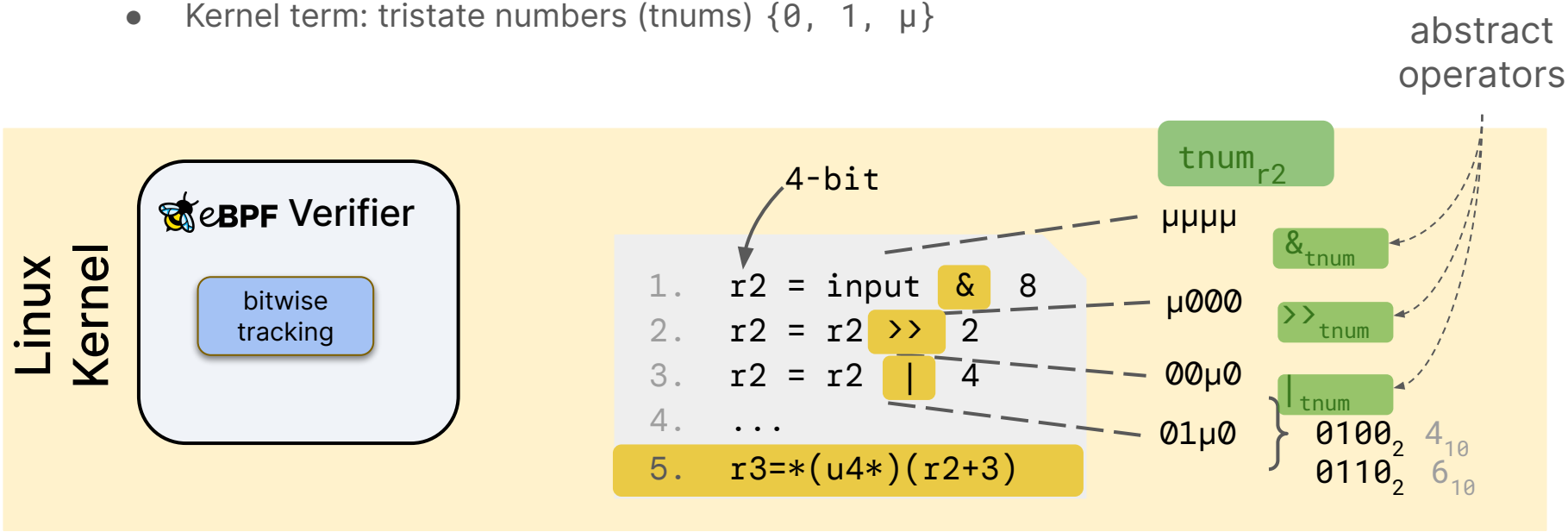
Static Analyses in the eBPF verifier



- Tracking the **values** of program variables across **all** executions of the program
- Our work: Reasoning about the soundness of the range analysis + bitwise tracking + their combination

Bitwise Tracking

- Task: track the **values** of program variables across **all** executions
  - Using abstract values from an abstract domain
- Bitwise domain: track **individual bits** of a program variable .
  - Kernel term: tristate numbers (tnums) {0, 1, μ}



Bitwise Tracking

- Task: track the **values** of program variables across **all** executions
  - Using abstract values from an abstract domain
- Bitwise domain: track **individual bits** of a program variable .
  - Kernel term: tristate numbers (tnums) {0, 1, μ}

```
1.  struct tnum tnum_and(struct tnum a, struct tnum b)
2.  {
3.      u6
4.      al
5.      be
6.      v = a.value & b.value;
7.      return TNUM(v, alpha & beta & ~v);
8.  }
```

Is tnum_and sound?

Constraints for a Sound Tnum Operator

$$
\begin{aligned}
&P, Q \in \texttt{tnums} \\
&x, y \in \texttt{integers}_{64} : \\
&\quad member(x, P) \quad \wedge \\
&\quad member(y, Q) \quad \wedge \\
&\quad z = x \;\&\; y \quad \wedge \\
&\quad R = P \;\&_{\texttt{tnum}}\; Q \quad \wedge \\
&\quad \neg member(z, R) \quad ?
\end{aligned}
$$

$$
\texttt{01µ0} \left\{ \begin{array}{ll} \texttt{0100}_2 & 4_{10} \\ \texttt{0110}_2 & 6_{10} \end{array} \right.
$$

$member(1, \texttt{01µ0}) = \texttt{false}$

$member(4, \texttt{01µ0}) = \texttt{true}$

Constraints for a Sound Tnum Operator

$$P, Q \in \texttt{tnums}$$

$$x, y \in \texttt{integers}_{64} :$$

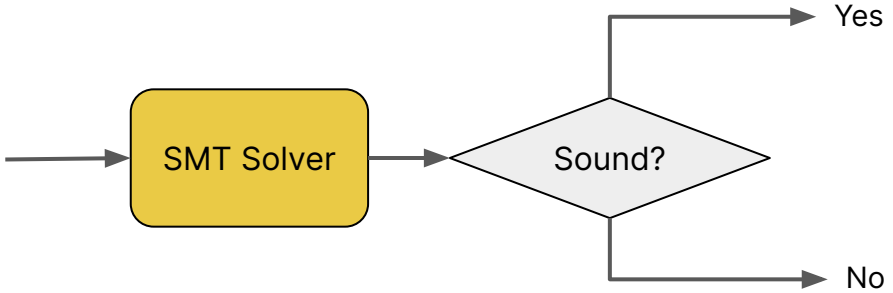
$$member(x, P) \quad \wedge$$

$$member(y, Q) \quad \wedge$$

$$z = x \;\&\; y \quad \wedge$$

$$R = P \;\&_{\texttt{tnum}}\; Q \quad \wedge$$

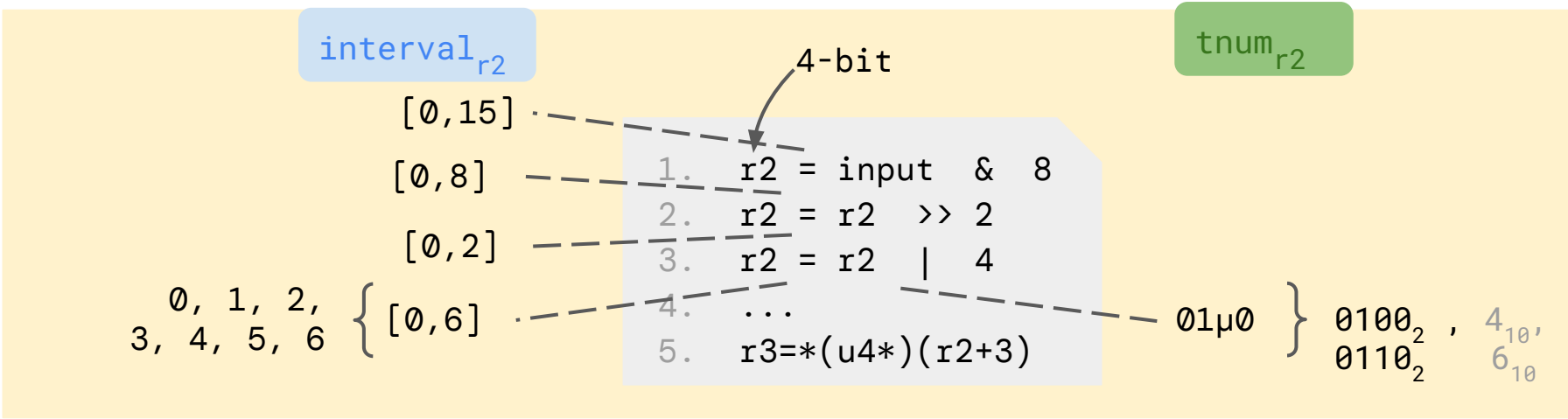
$$\neg member(z, R) \quad ?$$



Range Analysis

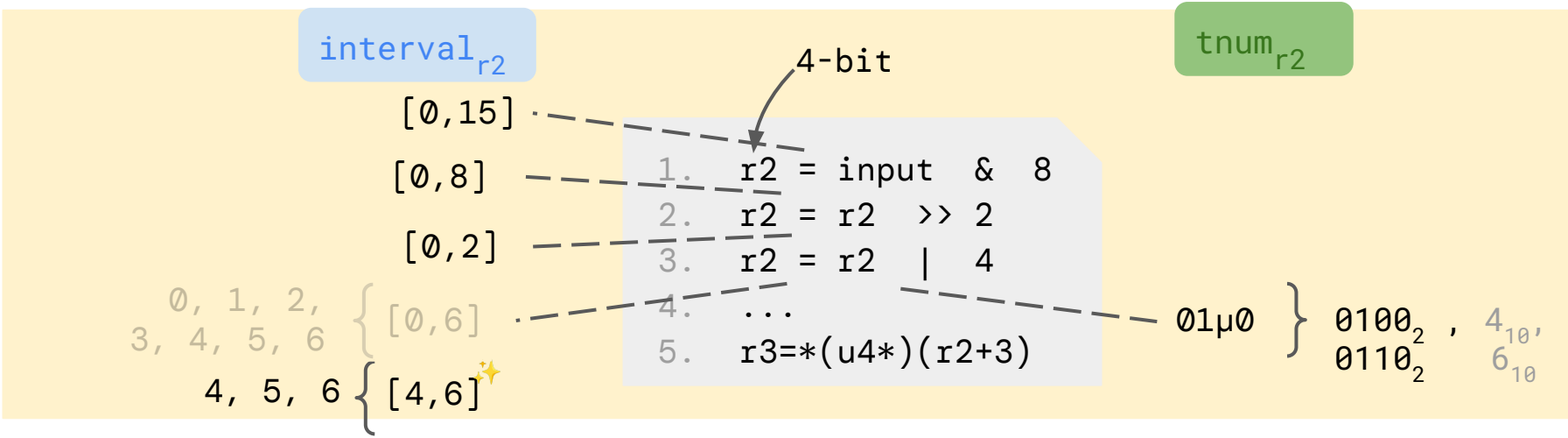
- Range Analysis: tracks range of possible values – [min, max]
  - Interval domain

Range Analysis



- Range Analysis: tracks range of possible values – [min, max]
  - Interval domain

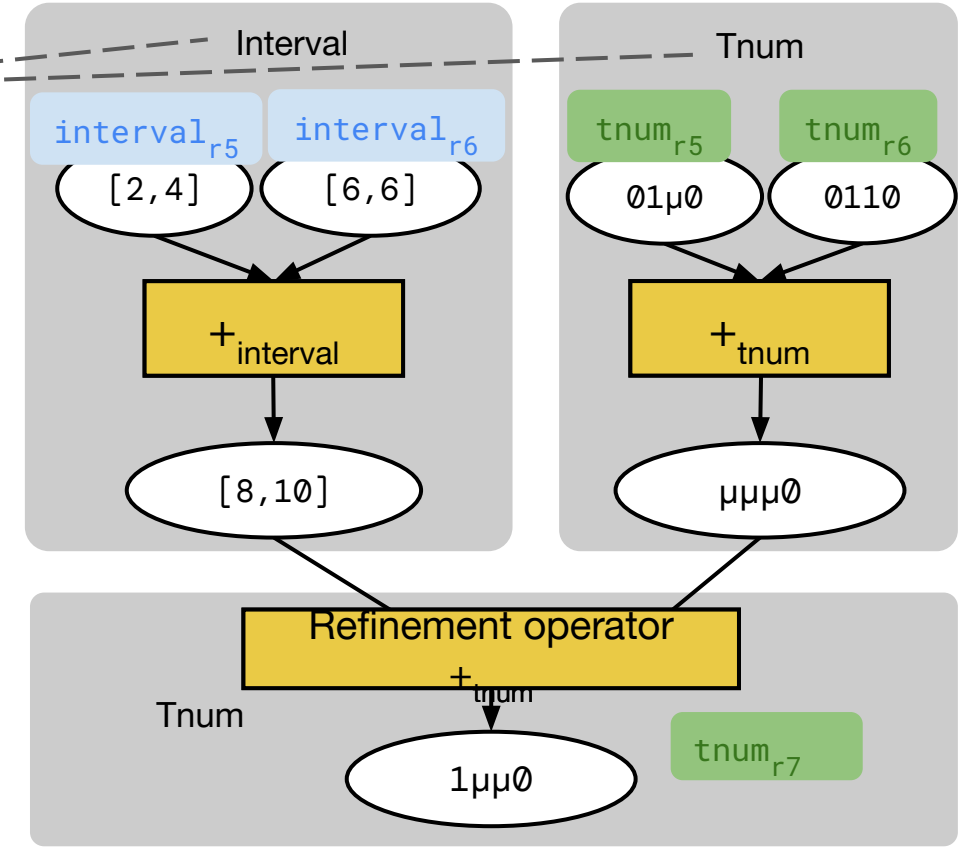
Range Analysis Refinement



- Range Analysis: tracks range of possible values – [min, max]
  - Interval domain
- Refinement: Abstract values in one domain can be used to refine abstract values in another domain

Modular Reasoning: The Usual Setting

```
42.   ...
43.   r7 = r5 + r6
44.   ...
```

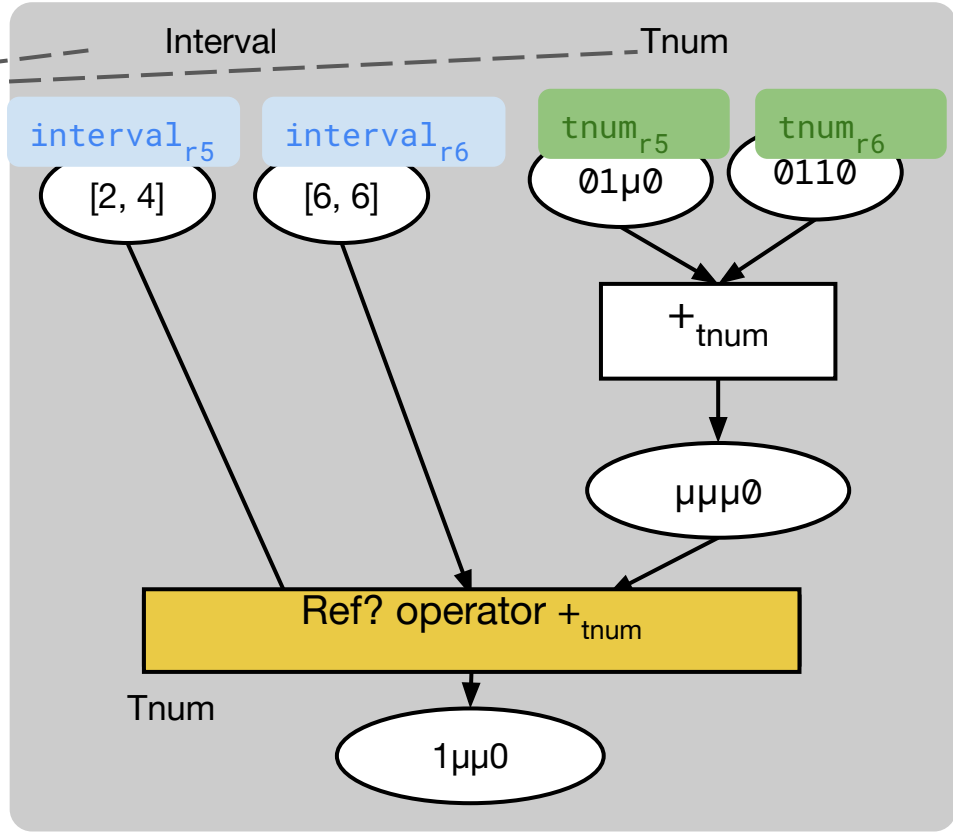


Soundness?

Modular reasoning

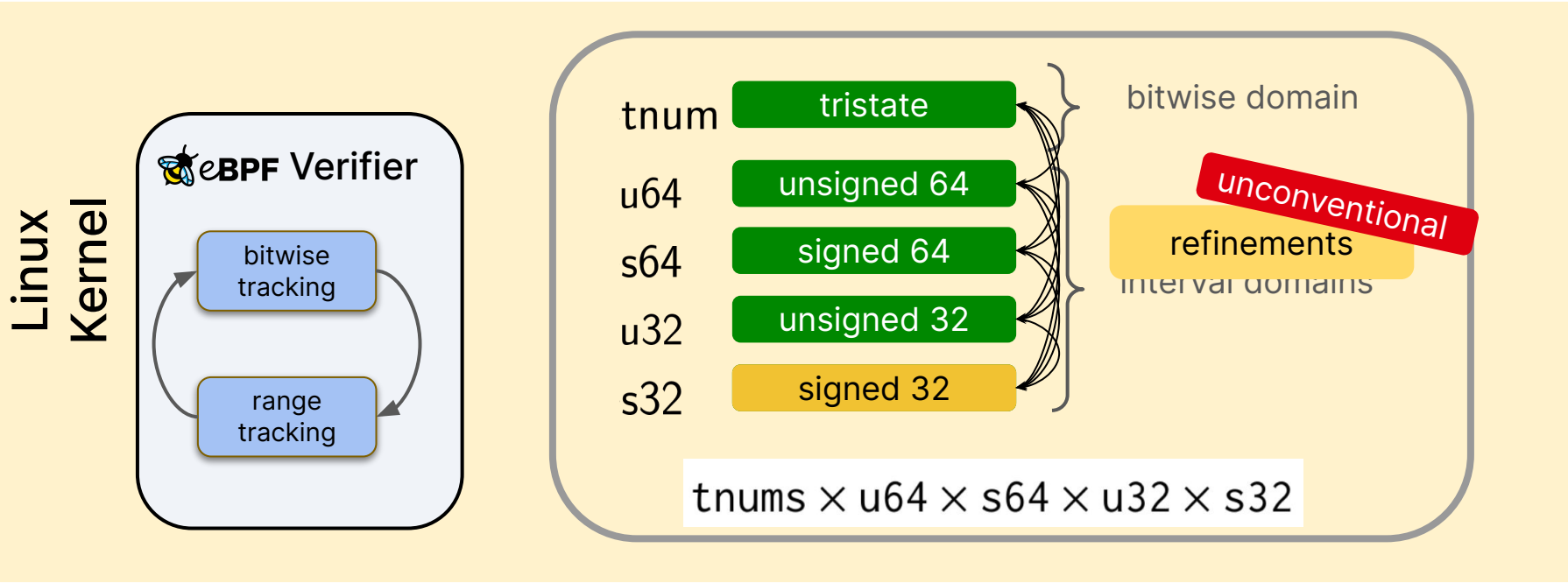
Non-Modular Reasoning: The eBPF Verifier

```
42.   ...
43.   r7 = r5 + r6
44.   ...
```



"One-shot" reasoning

Value Tracking Abstract Domains in the Linux Kernel



Constraints for Multi-Domain Soundness

$$
\begin{aligned}
&P, Q \in \texttt{tnums} \times \texttt{u64} \times \texttt{s64} \times \texttt{u32} \times \texttt{s32} : \\
&\quad x, y \in \texttt{integers}_{64} : \\
&\qquad member(x, P) \quad \wedge \\
&\qquad member(y, Q) \quad \wedge \\
&\qquad z = x \;\&\; y \quad \wedge \\
&\qquad R = P \;\&_{\texttt{tnums} \times \texttt{u64} \times \texttt{s64} \times \texttt{u32} \times \texttt{s32}}\; Q \quad \wedge \\
&\qquad \neg member(z, R)
\end{aligned}
$$

RUTGERS

Constraints for Multi-Domain Soundness

$$P, Q \in \text{tnums} \times \text{u64}$$

$$x, y \in \text{integers}_{64}$$

$$member(x, P)$$

$$member(y, Q)$$

$$z = x \;\&\; y$$

$$R = P \;\&_{\text{tnums} \times \text{u64} \times \text{s64} \times \text{u32} \times \text{s32}}\; Q$$

$$\neg member(z, R)$$

```
1.  static int adjust_scalar_min_max_vals(...)
2.  {
3.  switch (opcode) {
4.
5.    case BPF_AND:
6.       scalar_min_max_and(dst_reg, &src_reg);
7.       tnum_and(dst_reg->var_off, src_reg.var_off);
8.       break;
9.
10.   case BPF_SUB:
11.          scalar32_min_max_sub(dst_reg, &src_reg);
12. ...
```

$\wedge$

Scope of Automated Verification

- Arithmetic and Logic:
  - `add, sub, or, and, lsh,`
  - `div, mod` (Trivially sound)
  - `mul` ❌
- Jump:
  - `ja, jeq, jgt, jge, jlt, jle`
- Strict constraints: only report those than

Weaken constraints

true positive
✔ report

false positive
✘ do not report

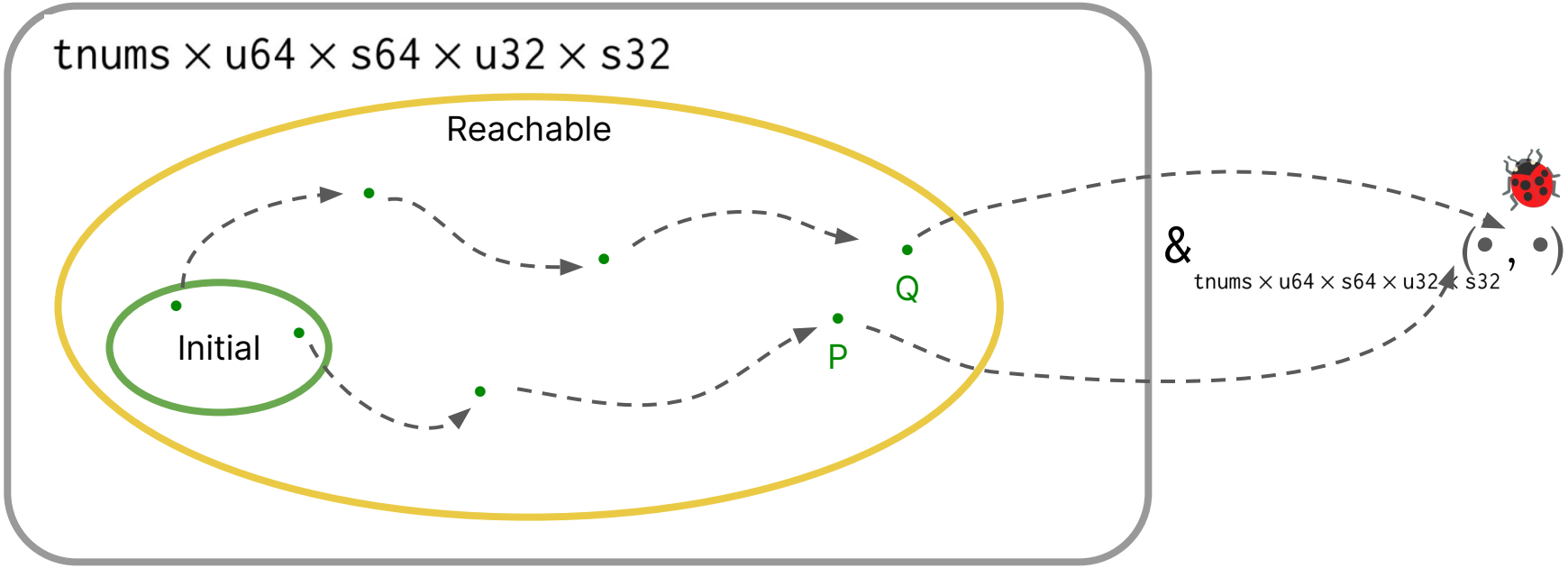
Results

- Proved that all abstract operators in kernels starting from v5.13 are sound
- What can we do about unsound versions?
  - False positive?
  - Are these bugs possible via real eBPF programs?

Generate *actual* eBPF programs!

Kernel Version	Sound?
v4.14	❌
v5.5	❌
v5.7	❌
...	❌
v5.12	❌
v5.13	✅
v5.14	✅
...	✅
v5.19	✅

Challenges in Synthesizing eBPF Programs that Manifest Bugs



Features of Agni's eBPF Program Synthesis

- Technique: A combination of an **enumerator** and a **solver**
- Supported instructions:
  - Arithmetic and Logic: `add, sub, or, and, lsh, rsh, xor, arsh`
  - Jump `ja, jeq, jgt, jge, jlt, jle, jset, jne, jsgt, jsge, jslt, jsle`

```
1.  r1 = ●
2.  r2 = ●
3.  r3 = ●
4.  if r1 > r2 goto +1
5.  exit
6.  r3 = r1 & r2
```

```
1.  r1 = ●
2.  r2 = ●
3.  r3 = r1 + r2
4.  r4 = r3 & ●
```

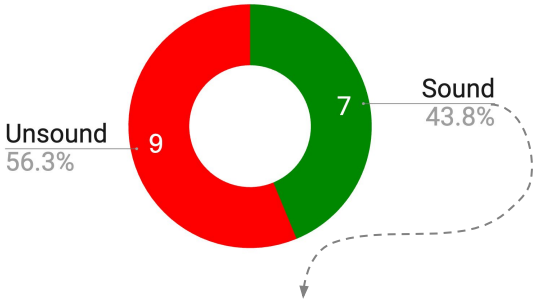
RUTGERS

Demo

Results

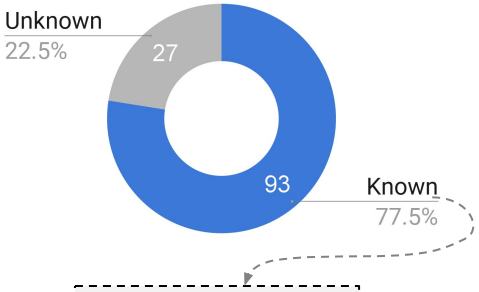
Agni: A verification tool for eBPF Range Analysis

Kernel Versions Tested



Proved versions 5.13 through 5.19 sound

Soundness Violations Across Kernel Versions



Part of 8 documented CVEs

eBPF Programs Synthesized

- Produced a POC for ~97% of soundness violations across kernel versions

Next Steps

- At a high-level: making Agni as push-button as possible
- Exploring the possibility of using Agni in Linux eBPF CI
- Reducing Verification Time
  - Initial work to parallelize verification of each instruction
- Reliable Code Extraction
- Lot's more to do!
- Possibilities for future verification:
  - Symbolic state pruning
  - eBPF helper function interface
- Ask me for a demo!

Hari harishankar.vishwanathan@rutgers.edu

Matan mys35@cs.rutgers.edu

Srinivas srinivas.narayana@rutgers.edu

Santosh santosh.nagarakatte@cs.rutgers.edu

https://github.com/bpfverif/agni