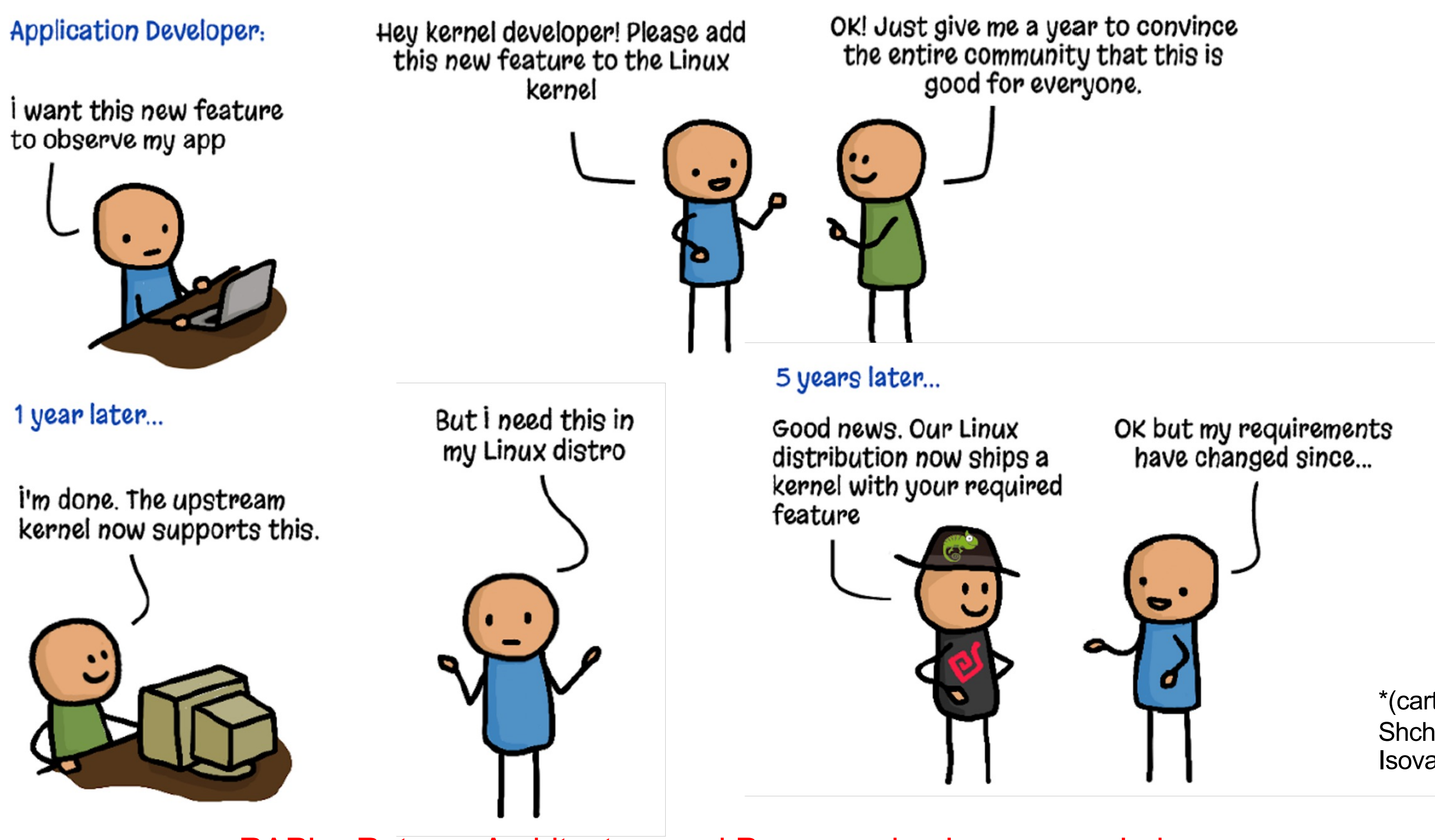
Fixing Latent Unsound Abstract Operators in the eBPF Verifier of the Linux Kernel

Santosh Nagarakatte @ SAS 2024,

Rutgers University

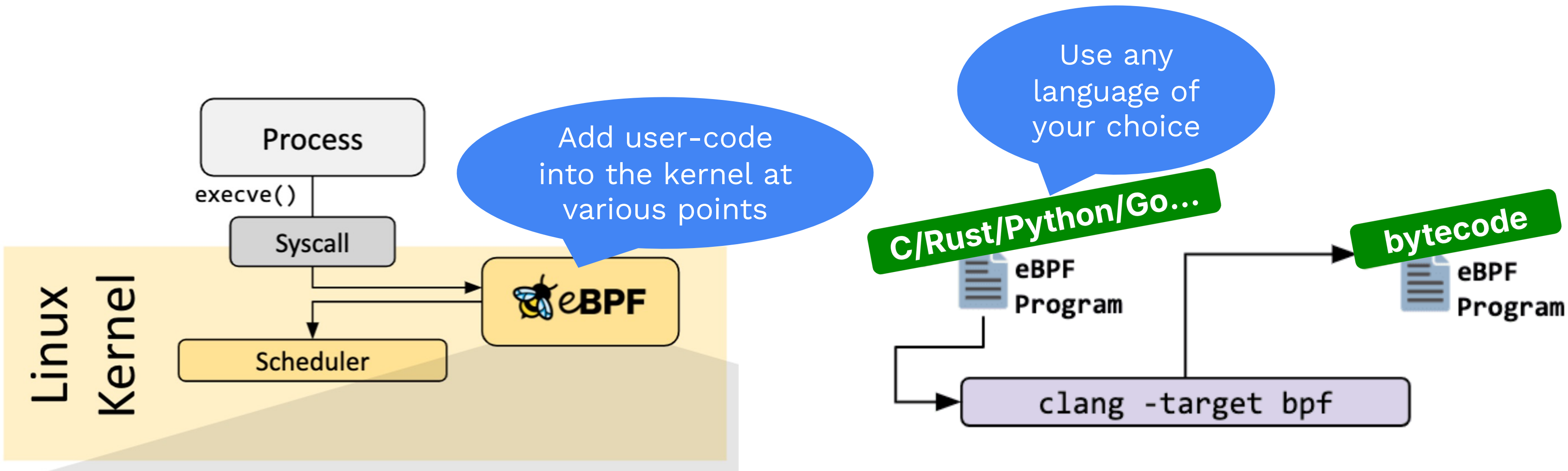
Joint work with Harishankar Vishwanathan, Matan Shachnai, and Srinivas Narayana

How to Extend the Functionality of the Linux Kernel?



*(cartoon by Vadim Shchekoldin, Isovalent)

The Extended Berkeley Packet Filter (eBPF)



Industry is Excited by the eBPF Ecosystem

How Netflix uses eBPF flow logs at scale for network insight

Netflix Technology Blog Follow
Jun 7, 2021 · 4 min read

By Alok Tiagi, Harihar
Lakshminarayan

Netflix has developed
that uses eBPF tracep
less than 1% of CPU a
sidecar provides flow

Making eBPF work on Winc

May 10, 2021 • 3 min read

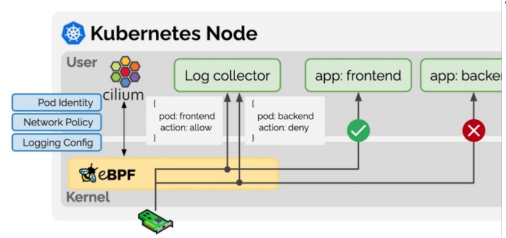
Dave Thaler
Partner Software Engineer, Microsoft

Poorna Gaddehosur
Principal Software Engineer Lead, Microsoft

eBPF is a well-known but revolutionary technology—provid
extensibility, and agility. eBPF has been applied to use case
protection and observability. Over time, a significant ecosystem of tools, products, and
experience has been built up around eBPF. Although support for eBPF was first
implemented in the Linux kernel, there has been increasing interest in allowing eBPF to
be used on other operating systems and also to extend user-mode services and
daemons in addition to just the kernel.

Using eBPF to build Kubernetes Network Policy Logging

Let's look at a concrete application of how eBPF is helping us solve a real customer
pain point. Security-conscious customers use Kubernetes network policies to
declare how pods can communicate with one another. However, there is no
scalable way to troubleshoot and audit the behavior of these policies, which r
it a non-starter for enterprise customers. With the introduction of eBPF to GK
can now support real-time policy enforcement as well as correlate policy acti
(allow/deny) to pod, namespace, and policy names at line rate with minimal i
on the node's CPU and memory resources.



Facebook, Google, Isovalent, Microsoft, and Netflix announce eBPF Foundation

‹ Back

eBPF Foundation

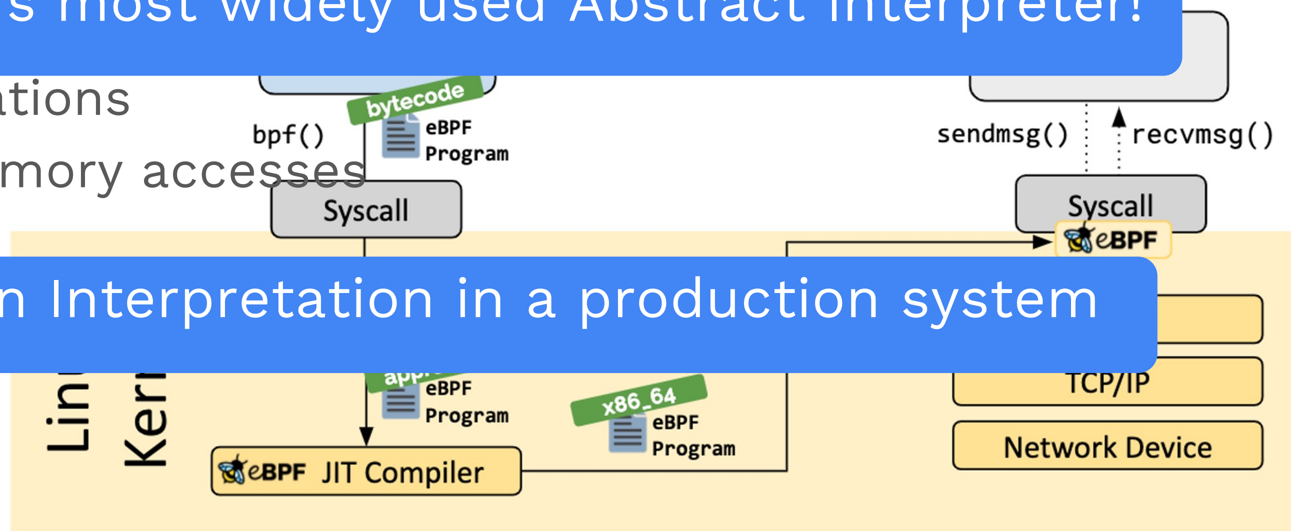
Founding Members

FACEBOOK Google ISOVALENT Microsoft NETFLIX

eBPF Verifier Guarantees Safety with Abstract Interpretation

- Running arbitrary user code in the kernel. Security?
- Kernel's solution: statically prove safety of the program
- Some properties to ensure safety
  - Termin
  - No illegal operations
  - Safe kernel memory accesses

World's most widely used Abstract Interpreter!

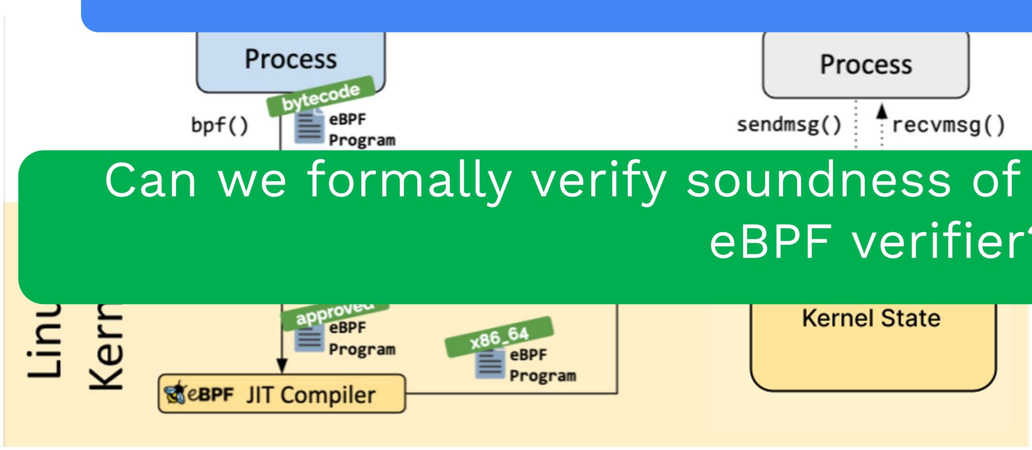


Abstraction Interpretation in a production system

The eBPF Verifier’s Goals: Be Sound, Precise, and Fast

- Soundness : Unsafe programs should be rejected
- Precision : Safe programs shouldn't be rejected
- Speed: Minimal load times + Prompt feedback on rejection

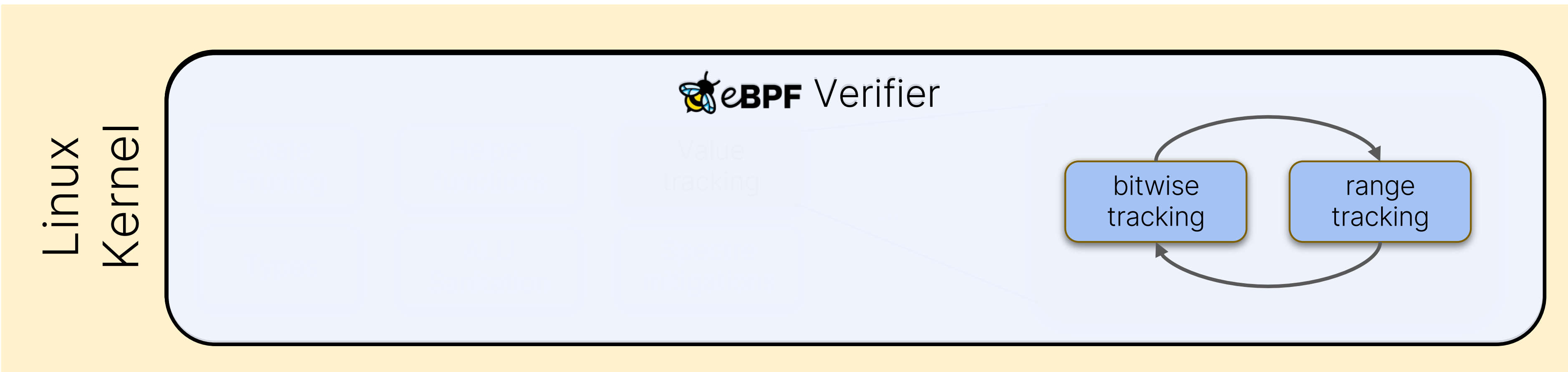
Writing sound and precise static analysis is hard



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

Images from https://ebpf.io/

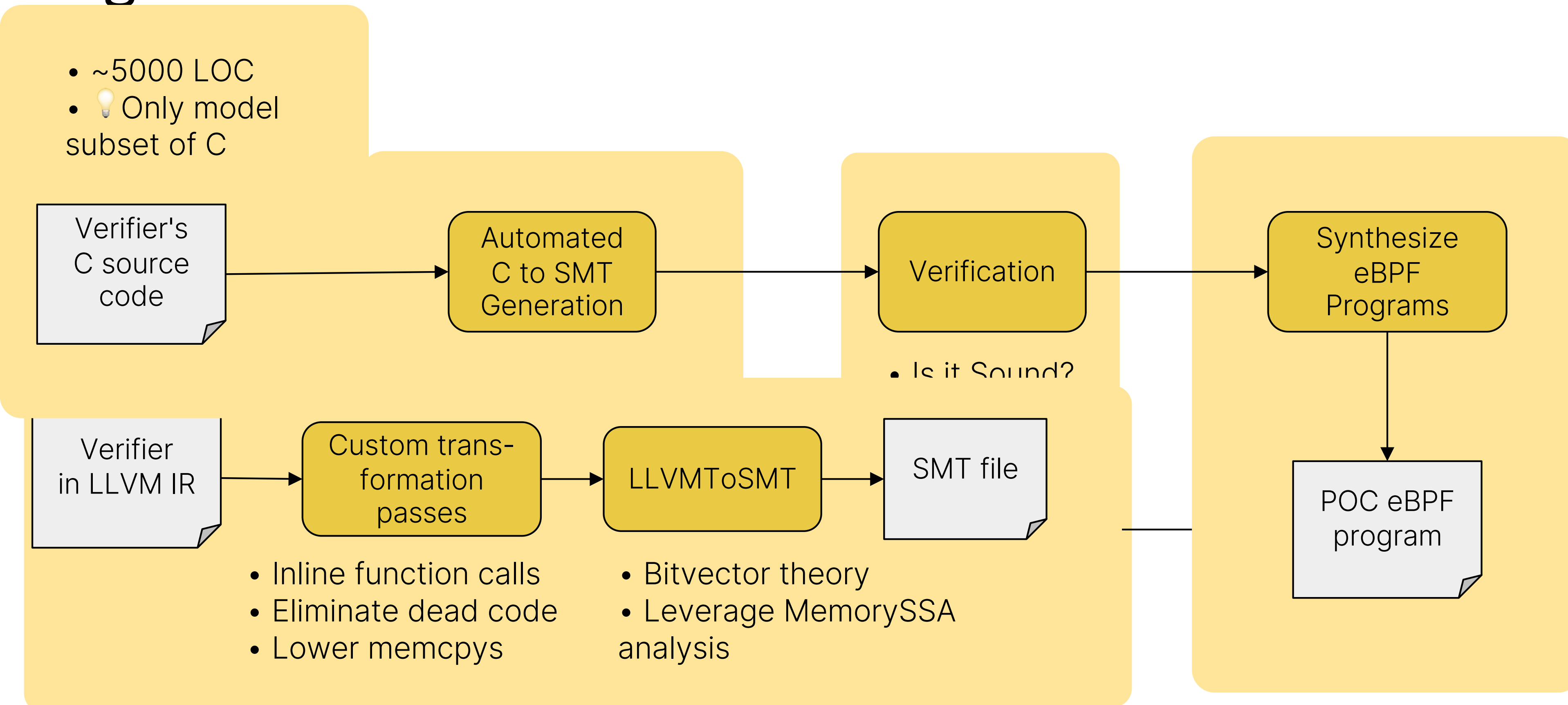
Static Analyses in the eBPF Verifier and Our Work



- **Tnums [CGO '22]**: Reasoning about the soundness of bitwise tracking – **Manually encoded** correctness specification and semi-manual verification
- **Agni [CAV '23]**: **Automated** reasoning about the soundness and precision of the range analysis + bitwise tracking + their combination
- **Agni++[SAS’24]**: [This Paper] Fixing the latent unsoundness in the abstract operators

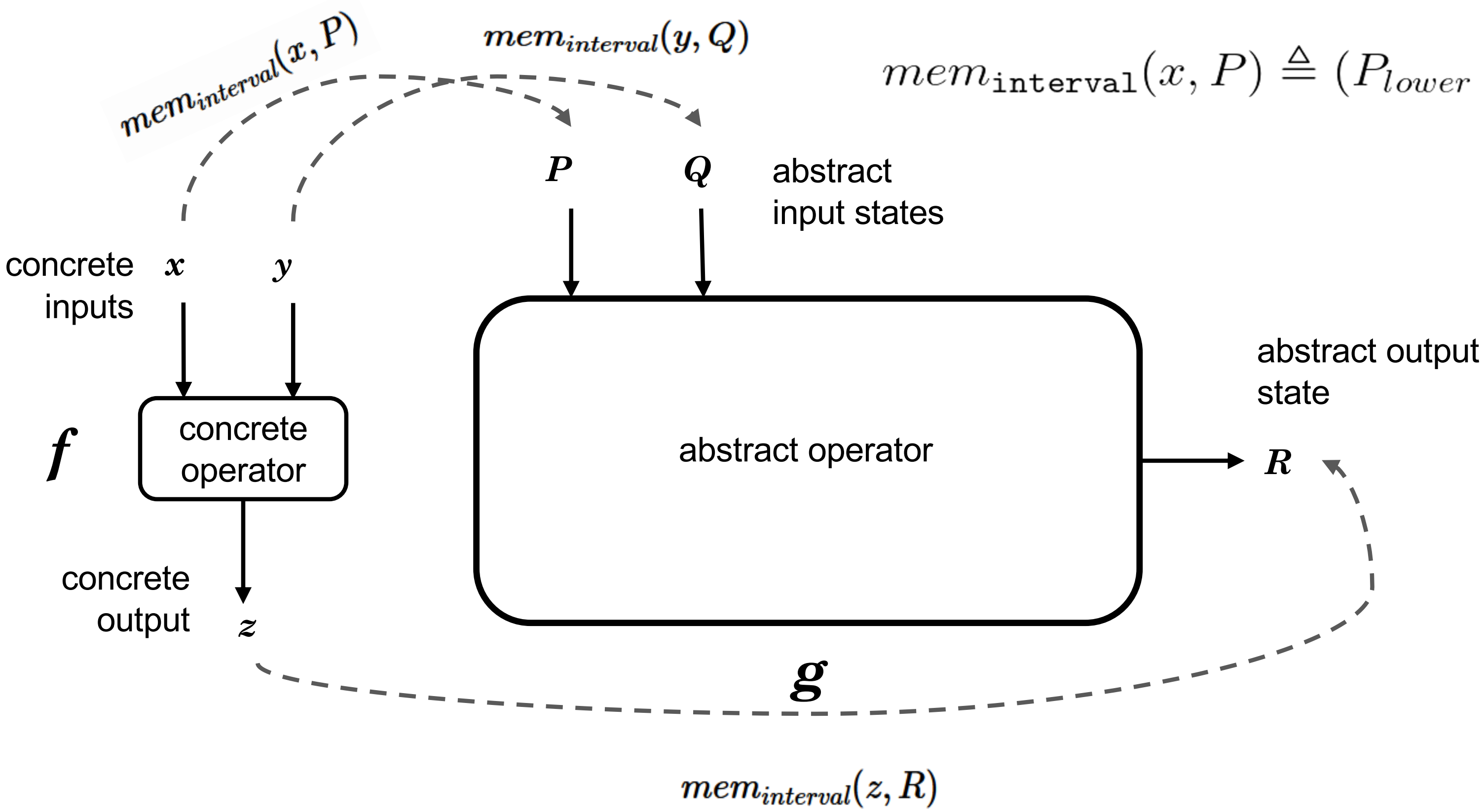
Develop **Automated Verification Tools** for use in the **Linux Kernel's Continuous Integration Testing Infrastructure**

Agni Verifier’s Overview - Part of Linux Kernel’s CI



When is an Abstract Operator Sound?

$$mem_{\texttt{interval}}(x, P) \triangleq (P_{lower} \leq x) \wedge (x \leq P_{upper})$$



Soundness Specification in First Order Logic

$$
\begin{aligned}
&\forall P, Q \in \mathbb{A}_{\texttt{interval}} : \\
&\forall x, y \in \mathbb{Z}_{\texttt{64}} : \\
&mem_{\texttt{interval}}(x, P) \ \wedge mem_{\texttt{interval}}(y, Q) \ \wedge \\
&z = f(x, y) \ \wedge \\
&R = g(P, Q) \implies mem_{\texttt{interval}}(z, R)
\end{aligned}
$$

Value Tracking Abstract Domains in the Linux Kernel

$\mathbb{A}_{\mathtt{t}}$	tristate	bitwise domain
$\mathbb{A}_{\mathtt{u64}}$	unsigned 64	interval domains
$\mathbb{A}_{\mathtt{s64}}$	signed 64	
$\mathbb{A}_{\mathtt{u32}}$	unsigned 32	
$\mathbb{A}_{\mathtt{s32}}$	signed 32	

Domain of all abstract domains: $\mathbb{A}$

$$\mathbb{A} \triangleq \mathbb{A}_{\mathtt{t}} \times \mathbb{A}_{\mathtt{u64}} \times \mathbb{A}_{\mathtt{s64}} \times \mathbb{A}_{\mathtt{u32}} \times \mathbb{A}_{\mathtt{s32}}$$

Soundness Specification with Multiple Domains

$$\forall P, Q \in \mathbb{A} :$$
$$\forall x, y \in \mathbb{Z}_{64} :$$
$$mem_{\mathbb{A}}(x, P) \wedge mem_{\mathbb{A}}(y, Q) \wedge$$
$$z = f(x, y) \wedge$$
$$R = g(P, Q) \implies mem_{\mathbb{A}}(z, R)$$

Challenges of Verifying Real World Code

- Performed verification on all kernel versions starting from v4.14
- Are all versions truly unsound?

What is the cause of verification failures?

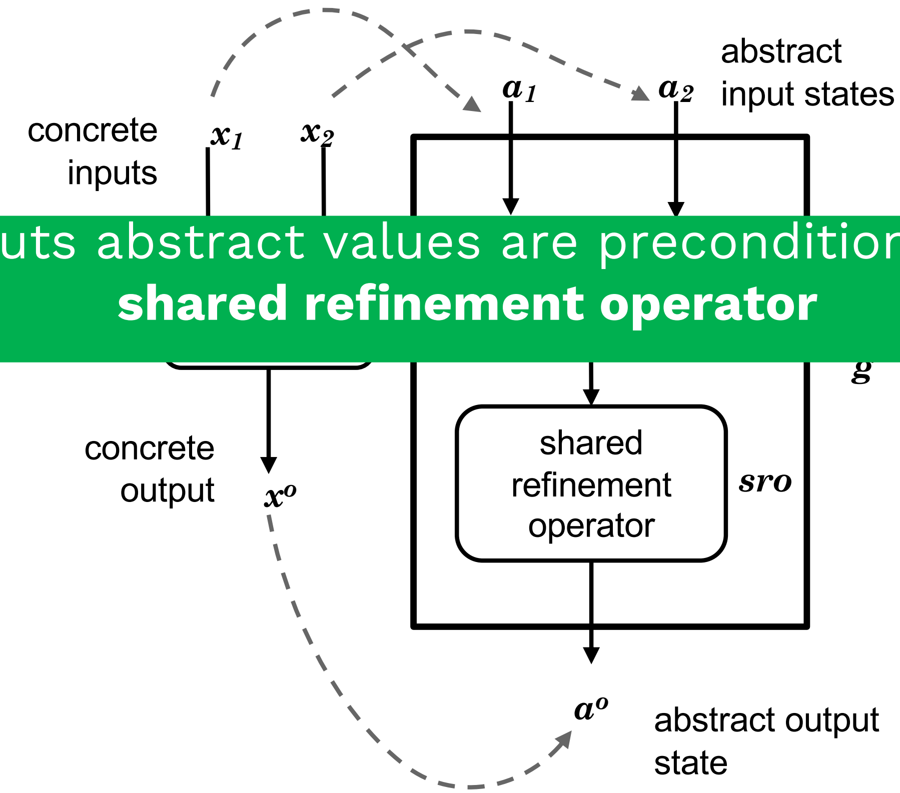
Kernel Version	Sound?
v4.14	✗
v5.5	✗
...	✗
v5.12	✗
v5.13	✗
v5.14	✗
v5.15	✗
...	✗

Implicit Refinement in the Kernel

```
abstract abstractALUOp(
    concreteOP op, abstract P, abstract Q)
{
    abstract R;
    switch (op) {
    case BPF_ADD:
        R = abstractOpADD(P, Q);
    case BPF_SUB:
        R = abstractOpSUB(P, Q);
     case BPF_MUL:
         R = abstractOpMUL(P, Q);
     .
     .
     .
     sync(R);
     return 0;
}
```

Shared refinement operator

Shared Refinement Operator Preconditions Abstract States



All inputs abstract values are preconditioned by a
shared refinement operator

A Soundness Specification in the presence of SRO

$$\forall P, Q \in \mathbb{A} :$$
$$\forall x, y \in \mathbb{Z}_{64} :$$
$$mem_{\mathbb{A}}(x, P) \ \wedge mem_{\mathbb{A}}(y, Q) \ \wedge$$
$$z = f(x, y) \ \wedge$$
$$R = g(P, Q) \implies mem_{\mathbb{A}}(z, R)$$

$$\forall P, Q \in \mathbb{A} :$$
$$R_p = sync(P) \ \wedge R_Q = sync(Q) \ \wedge$$
$$\forall x, y \in \mathbb{Z}_{64} :$$
$$mem_{\mathbb{A}}(x, R_P) \ \wedge mem_{\mathbb{A}}(y, R_Q) \ \wedge$$
$$z = f(x, y) \ \wedge$$
$$R = g(R_P, R_Q) \implies mem_{\mathbb{A}}(z, R)$$

Success in Proving the Soundness of Some Kernels

- Proved that all abstract operators in kernels starting from v5.13 are sound
- What can we do about unsound versions?

We generate *actual* eBPF programs using differential program synthesis! [see our CAV 2023 paper]

Kernel Version	Sound?
v4.14	❌
v5.5	❌
v5.7	❌
...	❌
v5.12	❌
v5.13	✅
v5.14	✅
v5.15	✅
...	✅

When Verification Tools are Continuously Used

author Alexei Starovoitov <ast@kernel.org> 2023-11-02 08:59:05 -0700
committer Alexei Starovoitov <ast@kernel.org> 2023-11-09 18:58:40 -0800
commit cd9c127069c040d6b022f1ff32fed4b52b9a4017 (patch)
tree 61c346feb4d979fc120c6802a38104f14f948551
parent bf4a64b9323f181df8aba32d66cb37b9fa5df959 (diff)
parent 4621202adc5bc0d1006af37fe8b9aca131387d3c (diff)
download bpf-next-cd9c127069c0.tar.gz

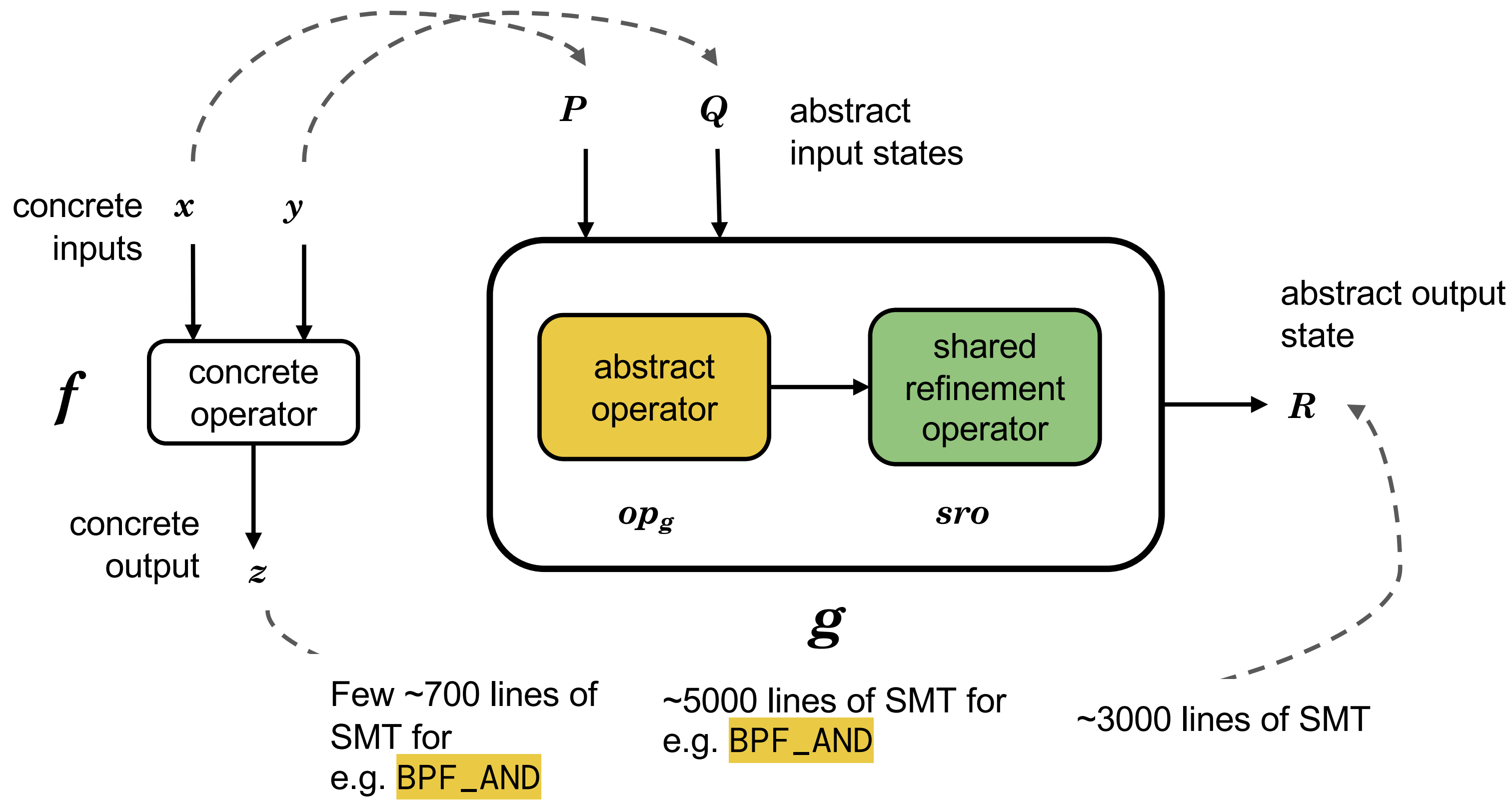
Merge branch 'bpf-register-bounds-logic-and-testing-improvements'

```
Andrii Nakryiko says:

====================
BPF register bounds logic and testing improvements

This patch set adds a big set of manual and auto-generated test cases
validating BPF verifier's register bounds tracking and deduction logic. See
details in the last patch.

We start w
verifier l
needed a b
covered. T
was incomp
implementation of register bounds logic that tests in this patch set
implement. So we need BPF verifier logic improvements to make all the tests
pass. This is what we do in patches #3 through #9.

The end goal of this work, though, is to extend BPF verifier range state
tracking such as to allow to derive new range bounds when comparing non-const
registers. There is some more investigative work required to investigate and
fix existing potential issues with range tracking as part of ALU/ALU64
operations, so <range> x <range> part of v5 patch set ([0]) is dropped until
these issues are sorted out.

For now, we include preparatory refactorings and clean ups, that set up BPF
verifier code base to extend the logic to <range> vs <range> logic in
subsequent patch set. Patches #10-#16 perform preliminary refactorings without
functionally changing anything. But they do clean up check_cond_jmp_op() logic
and generalize a bunch of other pieces in is_branch_taken() logic.

  [0] https://patchwork.kernel.org/project/netdevbpf/list/?series=797178&state=*

v5->v6:
  - dropped <range> vs <range> patches (original patches #18 through #23) to
    add more register range sanity checks and fix preexisting issues;
```

Kernel Version	Solving Time
v4.14	2.5h
v5.5	2.5h
v5.9	4h
v5.13	10h
[illegible]	[illegible]
v6.4	several weeks
v6.5	timeout
v6.6	timeout
v6.7	timeout
v6.8	timeout

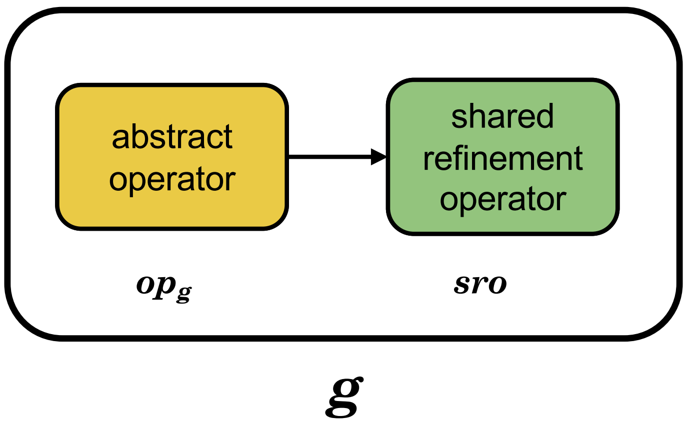
Can we significantly reduce the solving time?

Why is Solving Time Slow?

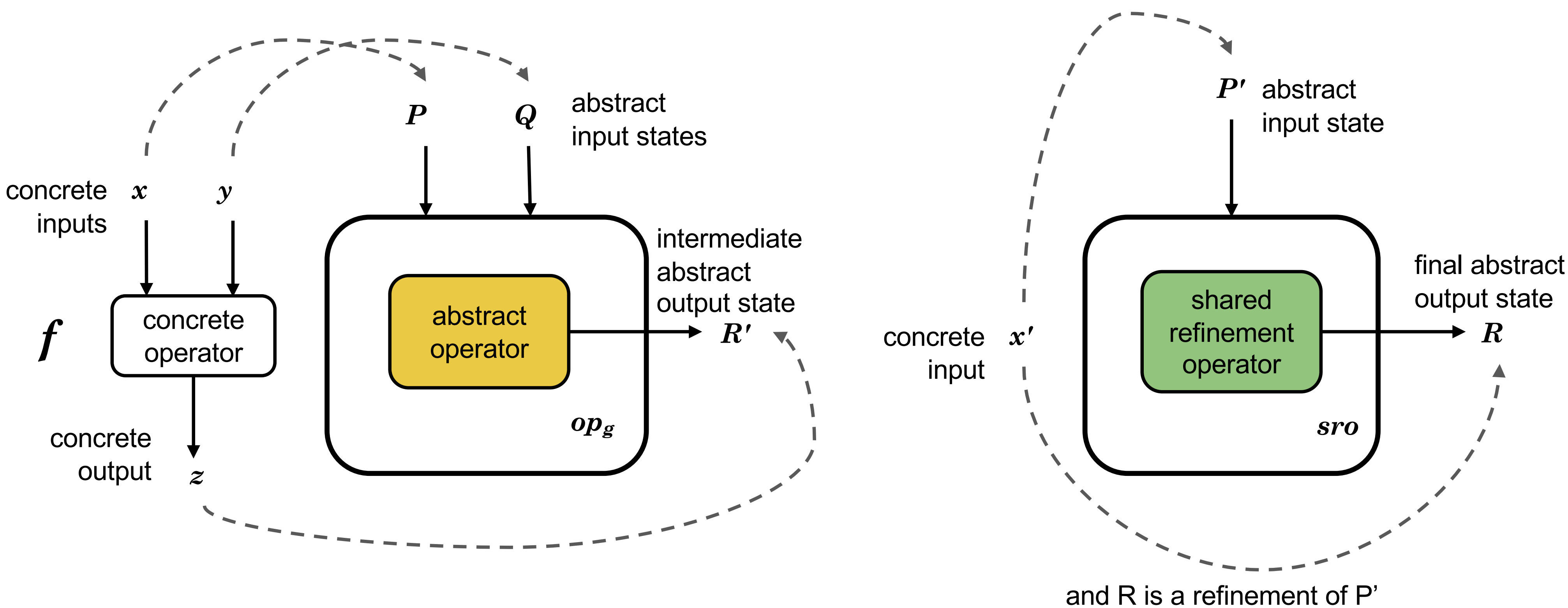


Divide and Conquer to Make Verification Feasible

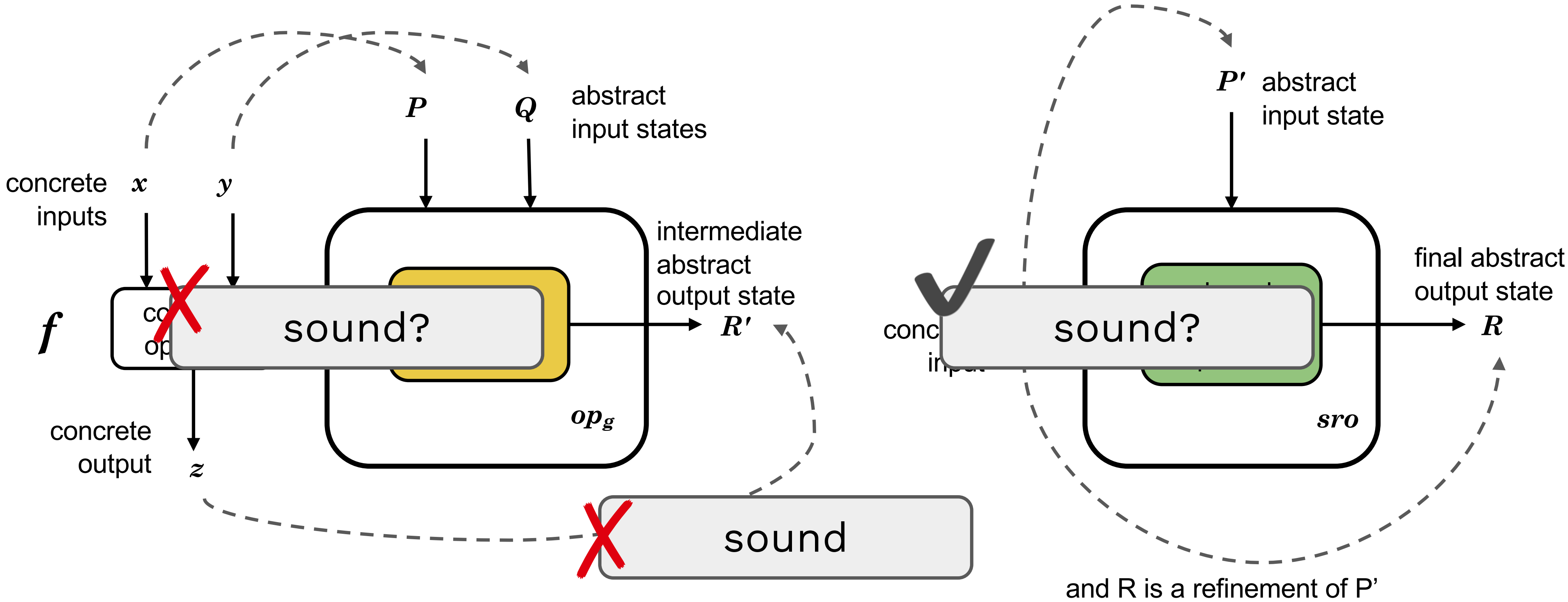
Can we individually verify $\boldsymbol{op_g}$ and $\boldsymbol{sro}$?



Divide and Conquer to Make Verification Feasible



Why Divide-and-Conquer Fails?



Latent Unsoundness in the Abstract Operators

```
case BPF_AND:
    out.tnum = tnum_and(in1, in2);
    out.s32, out.u32 = interval_and_32(in1, in2);
    out.s64, out.u64 = interval_and_64(in1, in2);
...
case BPF_OR:
        ...

out = sro(out);
```

Latent Unsoundness: interval_and_64

```
case BPF_AND:
      out.tnum = tnum_and(in1, in2);
      out.s32, out.u32 = interval_and_32(in1, in2);
      out.s64, out.u64 = interval_and_64(in1, in2);
```

Obtaining Signed Interval Bounds from Unsigned Interval Bounds!

```
def interval_and_64(in1, in2):
  out.u64_min = in1.tnum_value;
  out.u64_max = min(in1.u64_max, in2.u64_max);
  if (in1.s64_min < 0 || in2.s64_min < 0):
    out.s64_min = INT64_MIN;
    out.s64_max = INT64_MAX;
  else:
    out.s64_min = out.u64_min;
    out.s64_max = out.u64_max;
```

Unsafe casting - unsigned to signed

Avoiding Latent Unsoundness: When is such Casting Safe?

Unsafe casting - unsigned to signed

```
1. s64_min = u64_min;
2. s64_max = u64_max;
```

Unsigned		Signed
$u64\_min \leq u64\_max \leq 2^{63}-1$	✓ →	$0 \leq s64\_min \leq s64\_max$
$2^{63}-1 < u64\_min \leq u64\_max$	✓ →	$s64\_min \leq s64\_max < 0$
$u64\_min \leq 2^{63}-1 < u64\_max$	✗ →	$s64\_max < 0 \leq s64\_min$

Fixing Latent Unsoundness

```
def interval_and_64(in1, in2):
  …
  out.u64_min = in1.tnum_value;
  out.u64_max = min(in1.u64_max, in2.u64_max);
  if (in1.s64_min < 0 || in2.s64_min < 0):
    out.s64_min = INT64_MIN;
    out.s
  else:
    out.s6
    out.s64_max = out.u64_max;
      …
```

✗ sound?

Unsafe casting

```
def FIXED_interval_and_64(in1, in2):
  …
  out.u64_min = in1.tnum_value;
  out.u64_max = min(in1.u64_max, in2.u64_max);
  if ((s64) out.u64_min <= (s64) out.u64_max):
    out.s6_min = INT64_MIN;
    ou
  else:
    out.
    out.s64_max = out.u64_max;
      …
```

✓ sound?

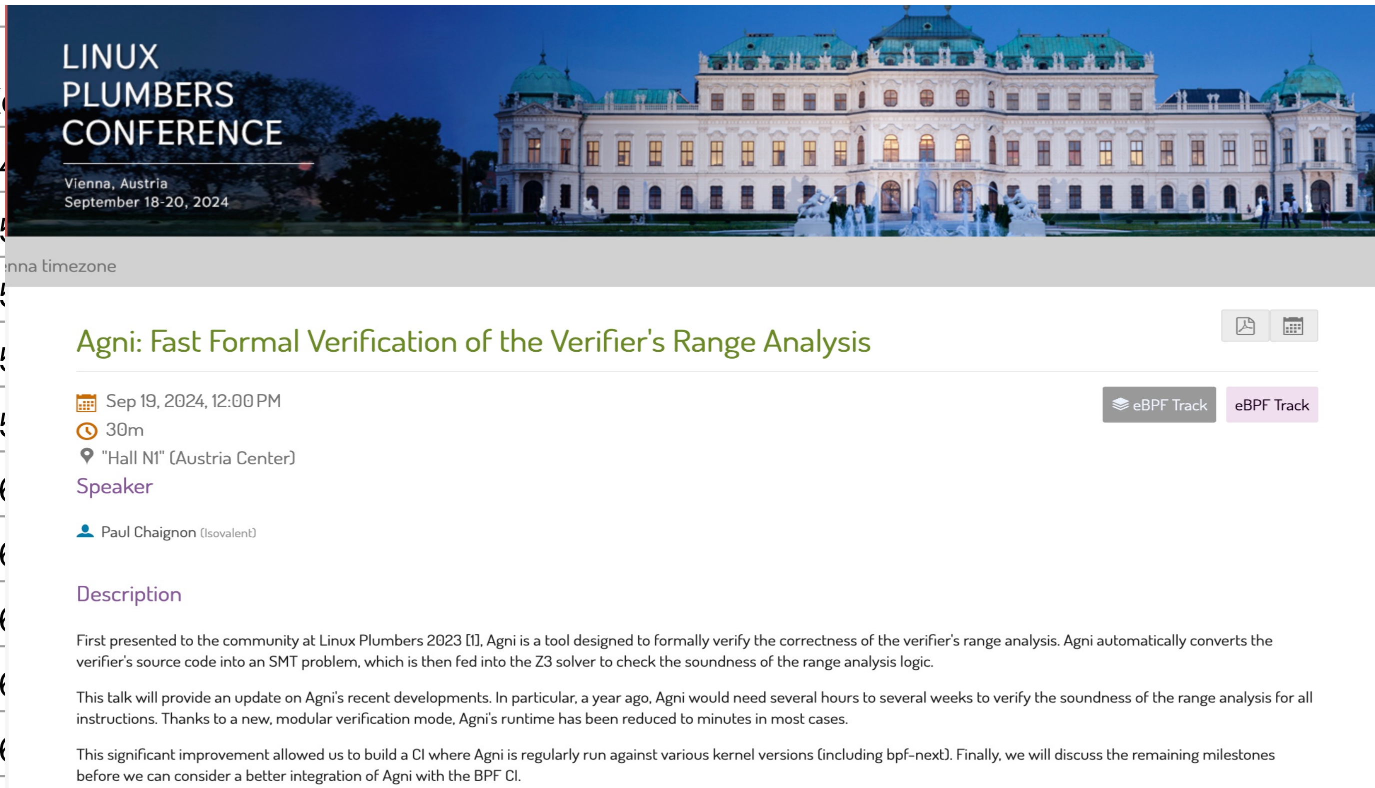
Safe casting

Divide-and-Conquer Makes Verification Super Fast!

Kernel Version	Old Strategy Runtime	New Strategy Runtime
v4.14	2.5h	<5 min
v5.5	2.5h	<5 min
v5.9	4h	<5 min
v5.13	10h	<5 min
v5.19	36h	<15 min
v6.3	36h	<15 min
v6.4	several weeks	<15 min
v6.5	timeout	<15 min
v6.6	timeout	<15 min
v6.7	timeout	<15 min
v6.8	timeout	<30 min

BPF Instruction	Sound before patch?	Sound after patch?
bpf_and	×	✓
bpf_and_32	×	✓
bpf_or	×	✓
bpf_or_32	×	✓
bpf_xor	×	✓
bpf_xor_32	×	✓

Divide-and-Conquer Makes Verification Super Fast!



K			…efore	Sound after patch?
v4				✓
v5				✓
v5				✓
v5				✓
v5				✓
v6				✓
v6				
v6				
v6				
v6				
v6.8	timeout	<30 min		

Real World Impact: Our Fixes Part of the Linux Kernel

```
author     Harishankar Vishwanathan <harishankar.vishwanathan@gmail.com>
committer     Daniel Borkmann <daniel@iogearbox.net>    2024-04-16 17:55:27 +0200
commit    1f586614f3ffa80fdf2116b2a1bebcdb5969cef8 (patch)
tree    7b5f4fa20fcbbdf316f4832c33d79dc8d4e8723d
parent    dac045fc9fa653e250f991ea8350b32cfec690d2 (diff)
download    bpf-next-1f586614f3ff.tar.gz
```

bpf: Harden and/or/xor value tracking in verifier

```
author
committe
commit
tree
parent
download
```

bpf-next: Avoid goto in regs_refine_cond_op()

```
author
commit
commit
tree
parent
downloa
```

bpf, tnums: Provably sound, faster, and more precise algorithm for tnum_mul

★ Integration of Agni into kernel CI - happening

Conclusion

- Kernel verification is hard but has real world value
- First steps to integrate formal methods into kernel development
- Some Linux Kernel developers are already using Agni
- Our ultimate goal: Verify the whole eBPF static analyzer

Open Source and Used by Linux Kernel Developers

Visit the Agni GitHub page for details: https://github.com/bpfverif/agni