

Formal Verification of the eBPF Verifier with Agni

Santosh Nagarakatte @ LSFMMBPF 2025, Montreal
Rutgers University

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



RAPL - Rutgers Architecture and Programming Languages Lab

eBPF Verifier's Goals: Soundness, Precision, and Speed



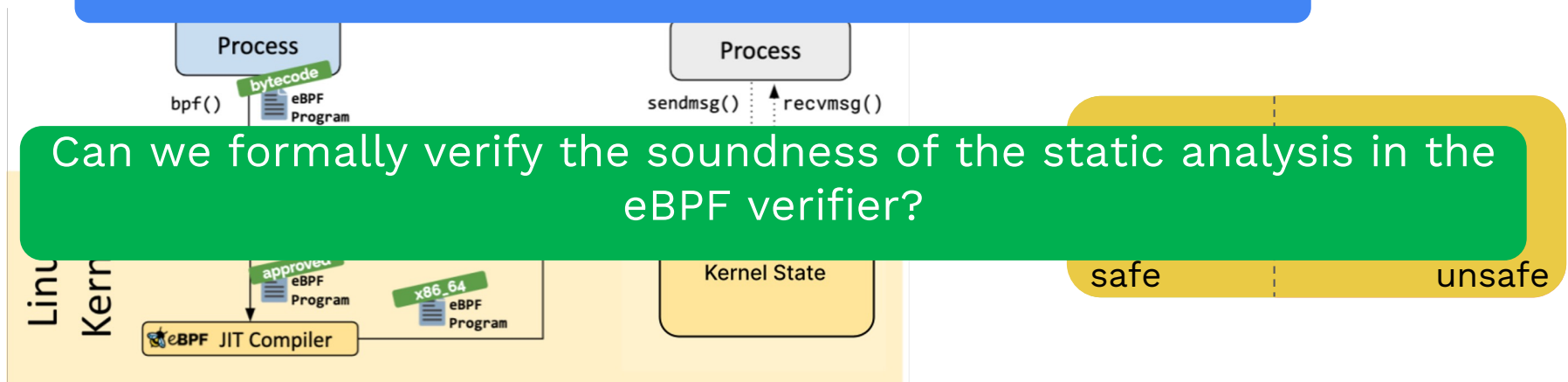
Soundness : Unsafe programs should be rejected



Precision : Safe programs shouldn't be rejected

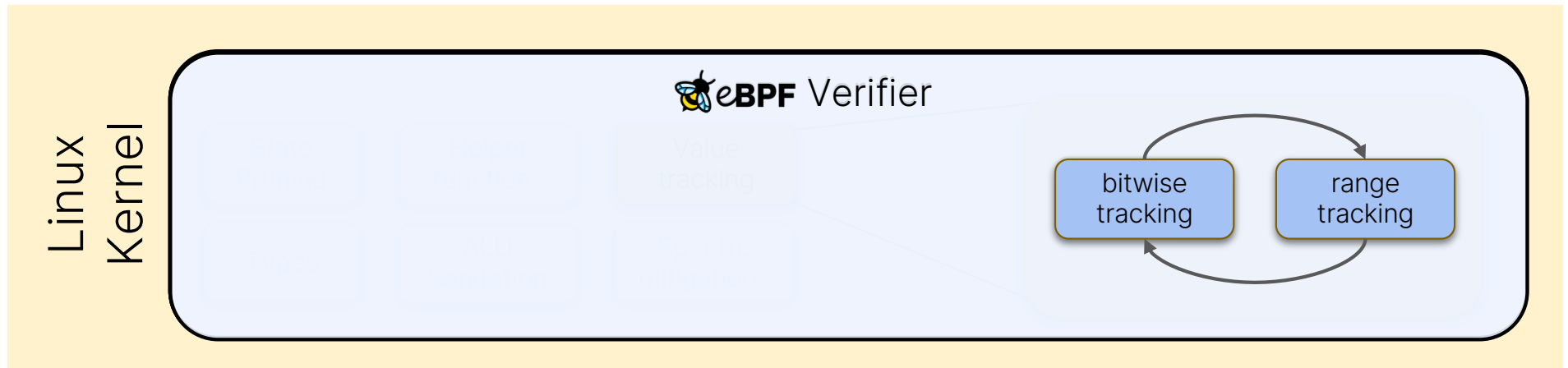
- Speed: Minimal load times + Prompt feedback on rejection

Writing sound and precise static analysis is hard



Images from <https://ebpf.io/>

Static Analyses in the eBPF Verifier and Our Work



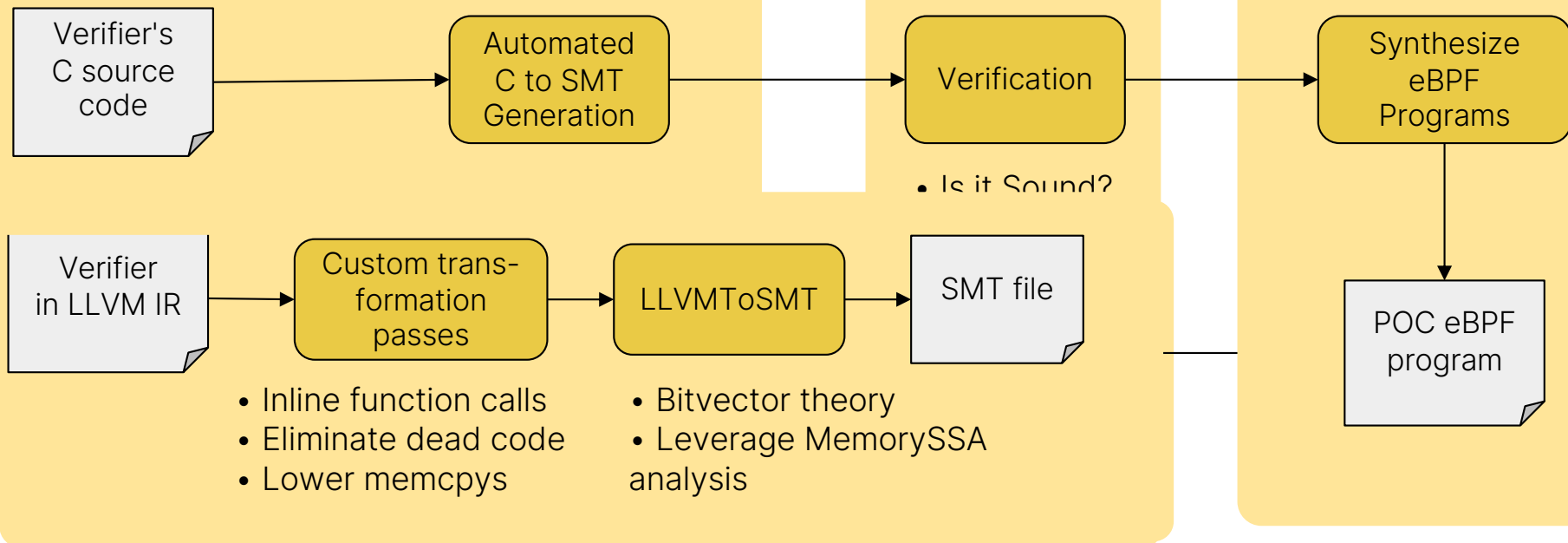
- **Tnums [CGO '22]**: Reasoning about the soundness of bitwise tracking – **Manually encoded** correctness specification and semi-manual verification
- **Agni [CAV '23]**: **Automated** reasoning about the soundness and precision of the range analysis + bitwise tracking + their combination
- **Agni++[SAS'24]**: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

Develop **Automated Verification Tools** that can be used to check a patch before it is accepted (e.g., as part of CI)

Thanks to Paul Chaignon for running CI with Agni for the latest bpf-next

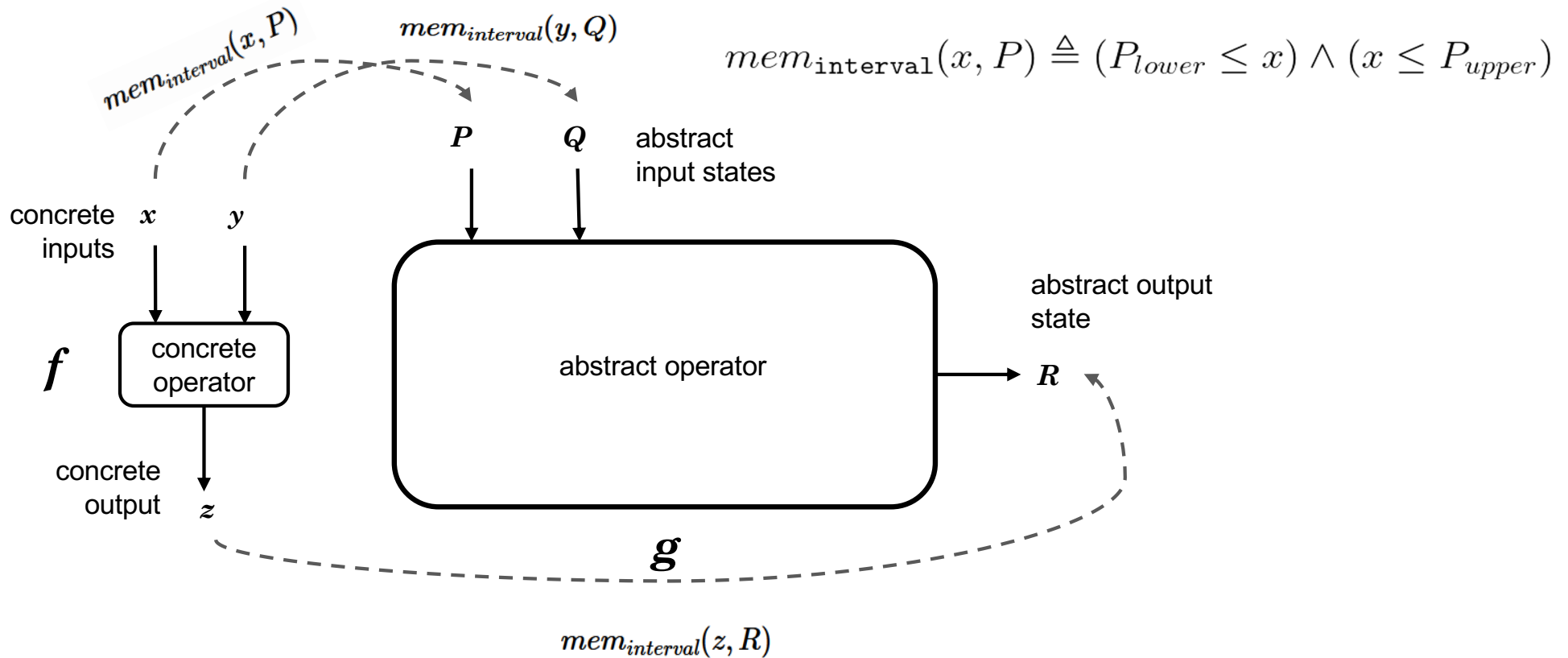
Overview of the Agni, which “Verifies the Verifier”

- ~5000 LOC
- 💡 Only model subset of C



**Specification at the lowest
abstraction level (C code) makes
verification challenging**

When is an Abstract Operator Sound?



Soundness Specification in First Order Logic

$\forall P, Q \in \mathbb{A}_{\text{interval}} :$

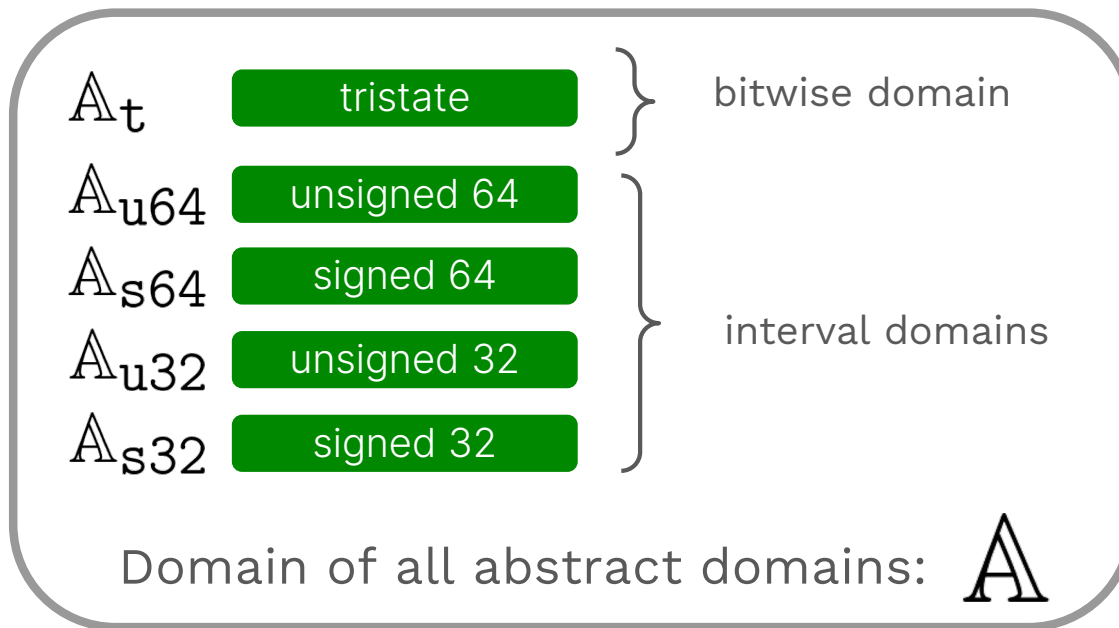
$\forall x, y \in \mathbb{Z}_{64} :$

$mem_{\text{interval}}(x, P) \wedge mem_{\text{interval}}(y, Q) \wedge$

$z = f(x, y) \wedge$

$R = g(P, Q) \implies mem_{\text{interval}}(z, R)$

Value Tracking Abstract Domains in the Linux Kernel



$$A \triangleq A_t \times A_{u64} \times A_{s64} \times A_{u32} \times A_{s32}$$

RAPL - Rutgers Architecture and Programming Languages Lab

Soundness Specification with Multiple Domains

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

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	✗
...	✗

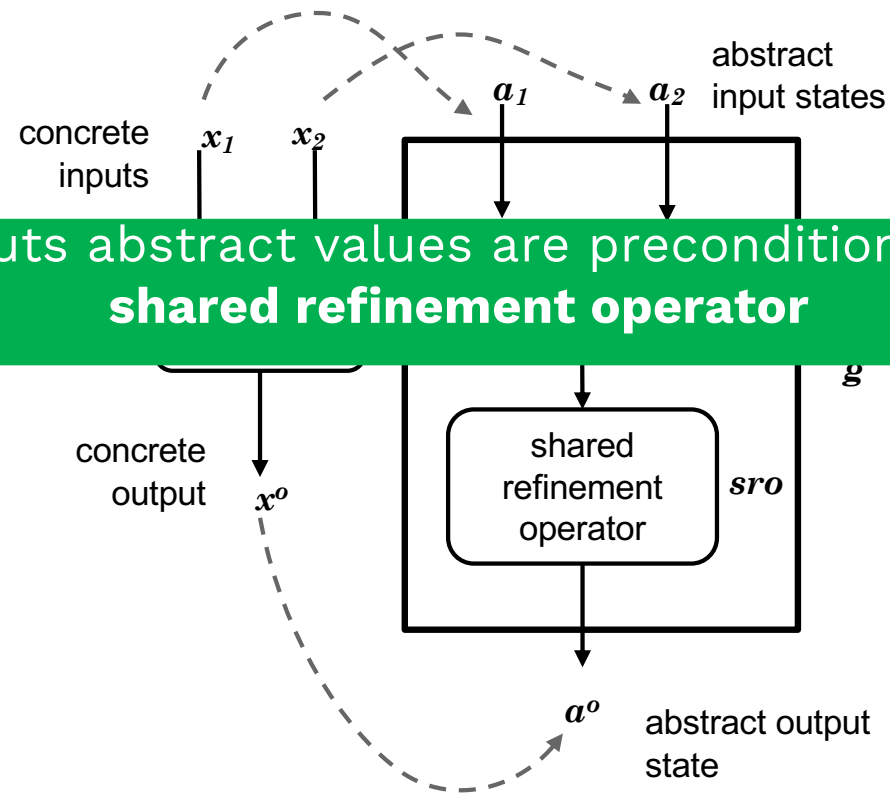
Implicit Refinement in the Kernel



```
1.abstract abstractALUOp(  
2.    concreteOP op, abstract P, abstract Q)  
3.{  
4.    abstract R;  
5.    switch (op) {  
6.    case BPF_ADD:  
7.        R = abstractOpADD(P, Q);  
8.    case BPF_SUB:  
9.        R = abstractOpSUB(P, Q);  
10.   case BPF_MUL:  
11.       R = abstractOpMUL(P, Q);  
12.   .  
13.   .  
14.   .  
15.   -- reg_bounds_sync(R);  
16.   return 0;  
17.}
```

Shared
refinement
operator

Shared Refinement Operator Preconditions Abstract States



A Soundness Specification in the presence of SRO

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

$$\begin{aligned} &\forall P, Q \in \mathbb{A} : \\ &\textcolor{blue}{R}_P = \textcolor{blue}{sync}(P) \wedge \textcolor{blue}{R}_Q = \textcolor{blue}{sync}(Q) \wedge \\ &\forall x, y \in \mathbb{Z}_{64} : \\ &\textcolor{red}{mem}_{\mathbb{A}}(x, \textcolor{blue}{R}_P) \wedge \textcolor{red}{mem}_{\mathbb{A}}(y, \textcolor{blue}{R}_Q) \wedge \\ &z = f(x, y) \wedge \\ &R = g(\textcolor{blue}{R}_P, \textcolor{blue}{R}_Q) \implies \textcolor{red}{mem}_{\mathbb{A}}(z, R) \end{aligned}$$

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?

How do we convince developers that these actual bugs?

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

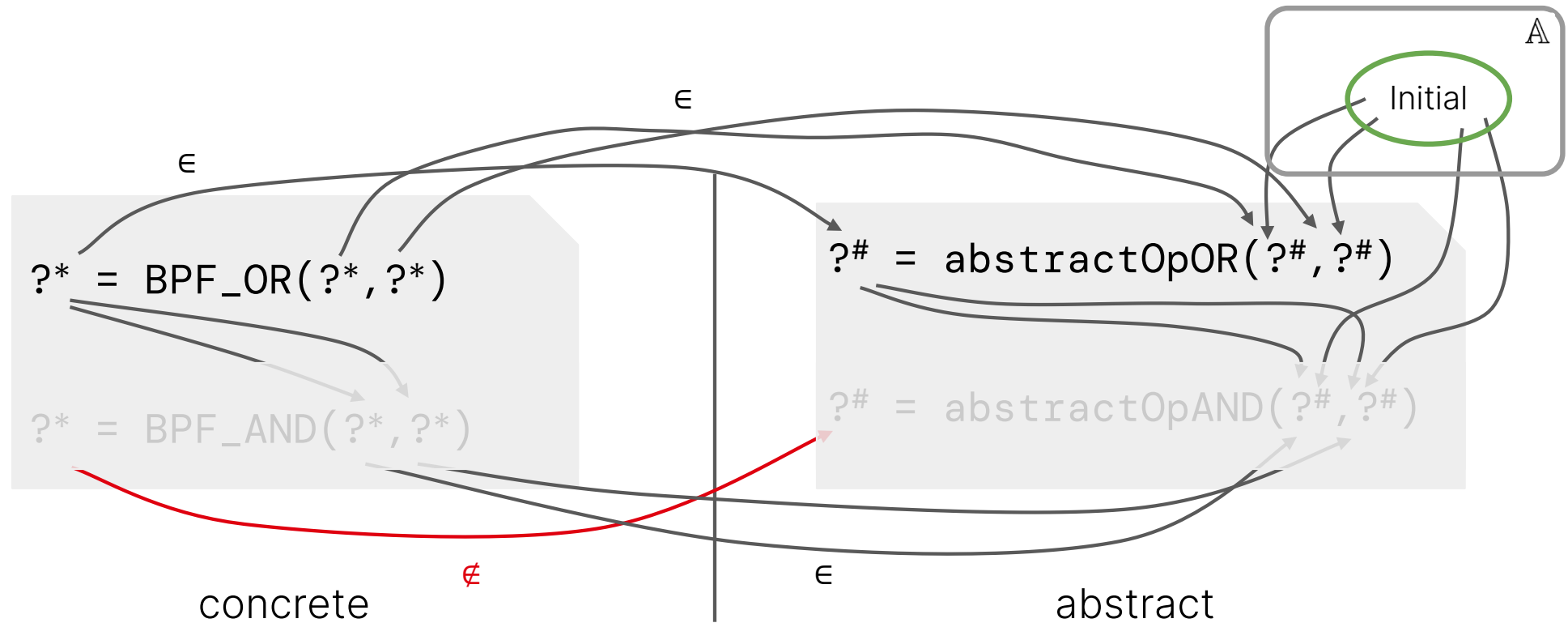
Kernel Version	Sound?
v4.14	✗
v5.5	✗
v5.7	✗
...	✗
v5.12	✗
v5.13	✓
v5.14	✓
v5.15	✓
...	✓

**What does an operator being
unsound mean?**

**There exists input abstract states
where the operator produces an ill-
formed output abstract state**

Is this “Unsoundness” realizable?

Differential Synthesis for Synthesizing eBPF Programs



**Concrete Proof of Concept
Programs were helpful to
reproducing the bugs**

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 with a BPF verifier test suite that needed a bit of work to be covered. The current implementation of register bounds logic that tests in this patch set is incomplete. 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_jump_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=*

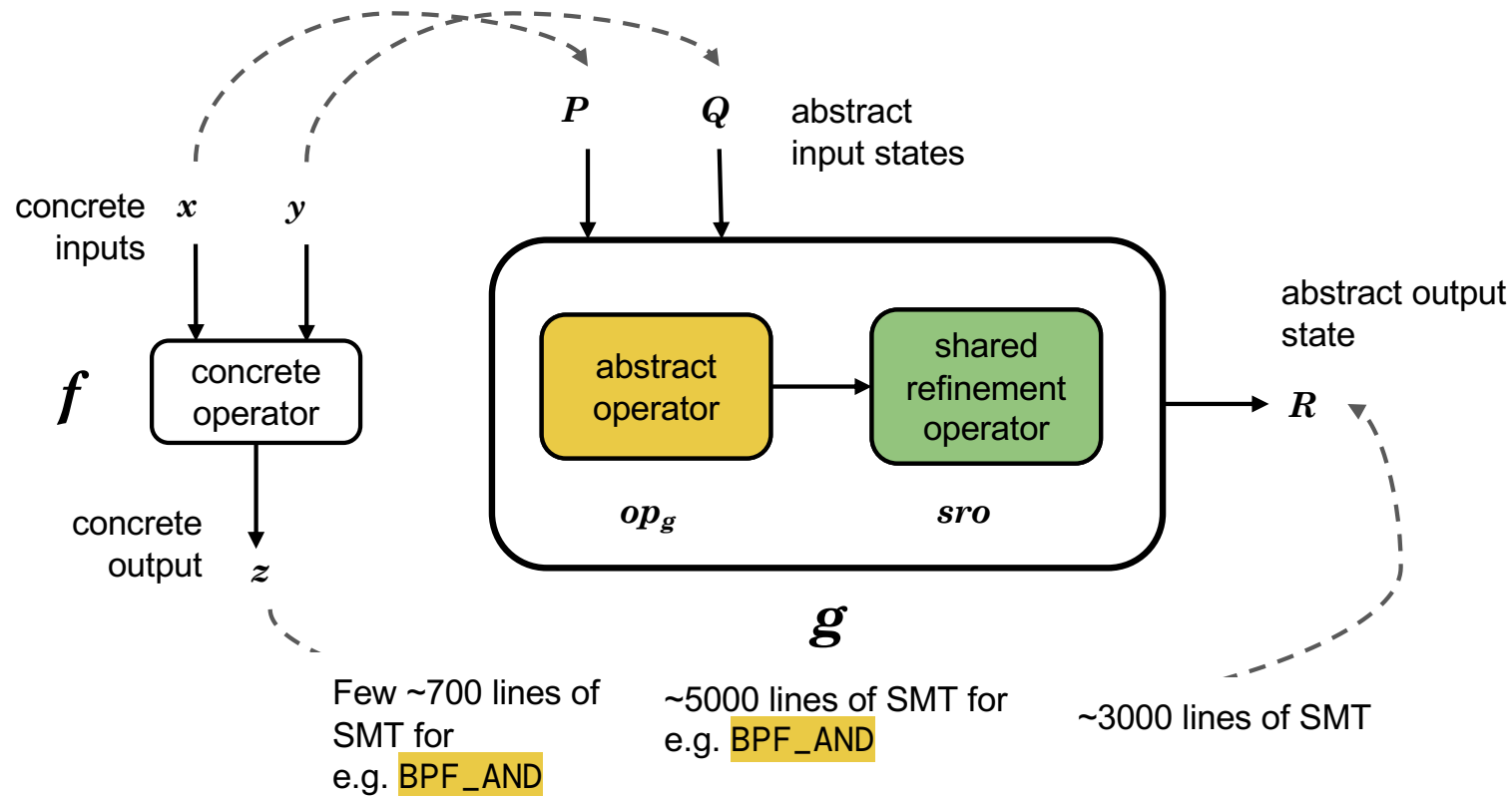
v5->v6:

- 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
v6.0	20h
v6.4	several weeks
v6.5	timeout
v6.6	timeout
v6.7	timeout
v6.8	timeout

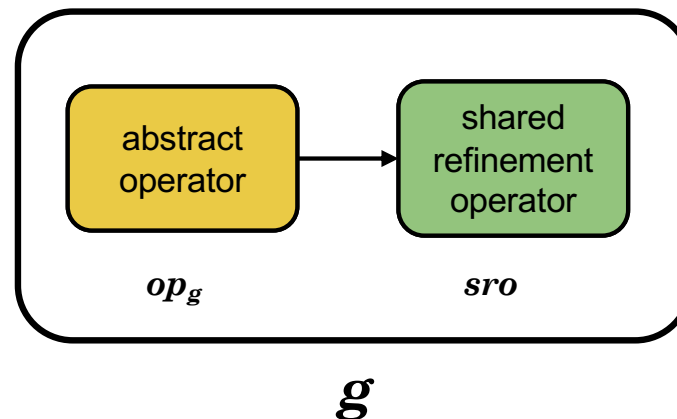
Can we significantly reduce the solving time?

Why is Solving Time Slow?

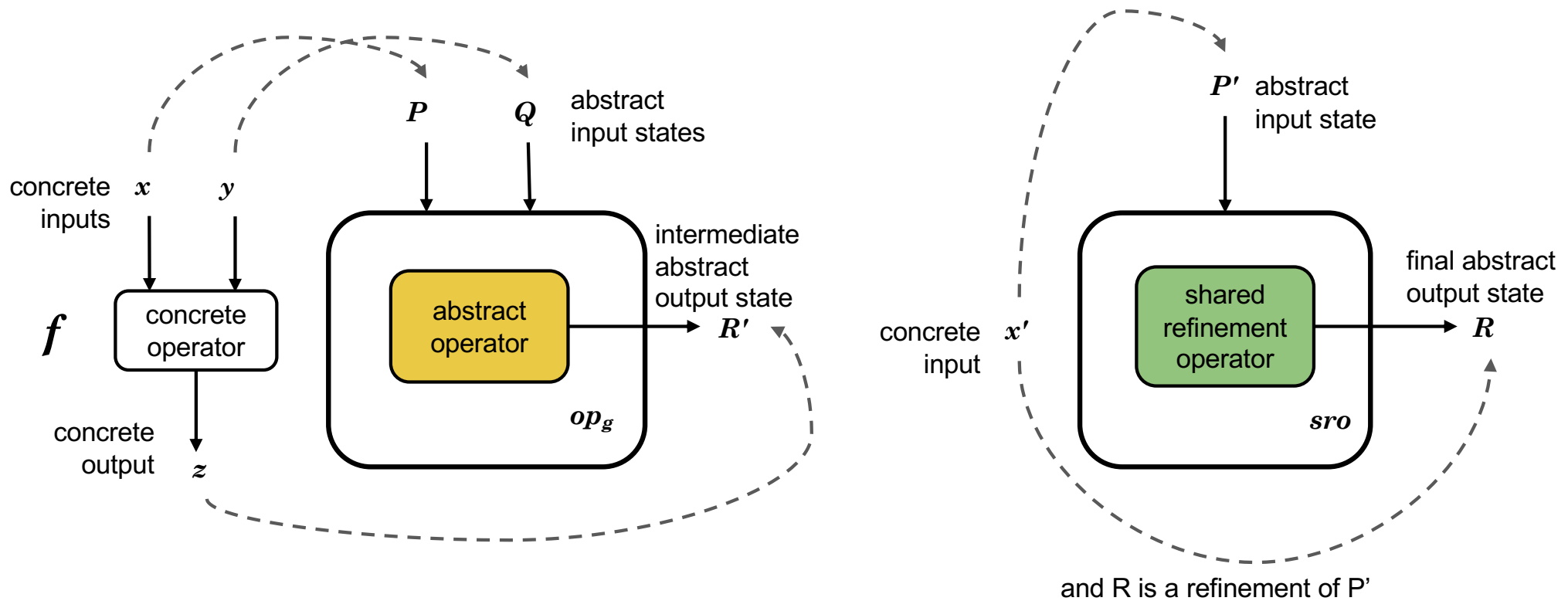


Divide and Conquer to Make Verification Feasible

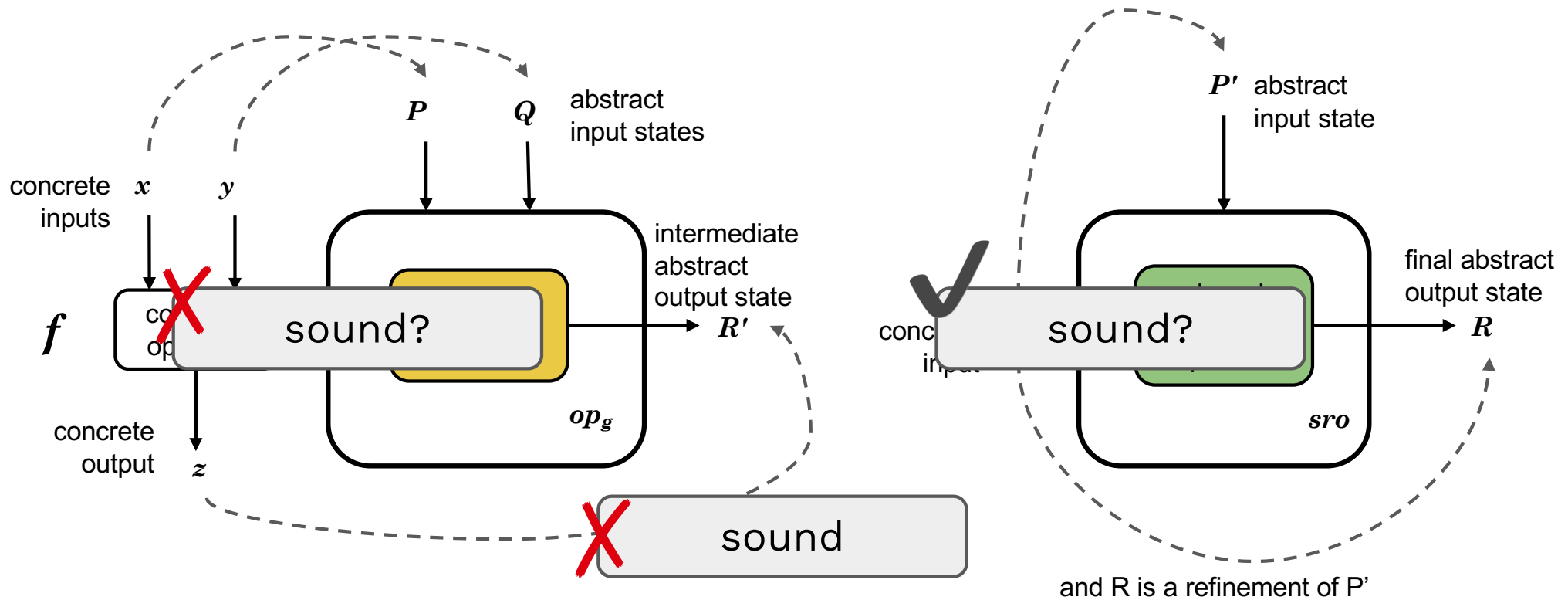
Can we individually verify op_g and sro ?



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!

```
1. def interval_and_64(in1, in2):
2.     out.u64_min = in1.tnum_value;
3.     out.u64_max = min(in1.u64_max, in2.u64_max);
4.     if (in1.s64_min < 0 || in2.s64_min < 0):
5.         out.s64_min = INT64_MIN;
6.         out.s64_max = INT64_MAX;
7.     else:
8.         out.s64_min = out.u64_min;
9.         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.s64_max = INT64_MAX;  
    else:  
        out.s64_min = in1.s64_min;  
        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.s64_min = INT64_MIN;  
        out.s64_max = INT64_MAX;  
    else:  
        out.s64_min = in1.s64_min;  
        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	✗	✓

Some Patches Upstreamed after Verification with Agni

```
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

author      Harishankar Vishwanathan <harishankar.vishwanathan@gmail.com>
committer   Daniel Borkmann <daniel@iogearbox.net>    2024-04-16 17:55:27 +0200
commit      05924717ac70e8e0f0f484780d7b90a63ea50020ac4bb027178d (diff)
tree        7dc40333e8e0f0f484780d7b90a63ea50020ac4bb027178d
parent      e8e0f0f484780d7b90a63ea50020ac4bb027178d (diff)
download    bpf-next-05924717ac70.tar.gz
```

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

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



Running Agni as part of CI – Thanks Paul and Hari

Improving the Precision of the Abstract Operators

```
* [PATCH bpf-next v4 1/2] bpf, verifier: Improve precision of BPF_MUL
2024-12-18 3:23 [PATCH bpf-next v4 0/2] bpf, verifier: Improve precision of BPF_MUL Matan Shachnai
@ 2024-12-18 3:23 ` Matan Shachnai
2024-12-18 3:23 ` [PATCH bpf-next v4 2/2] selftests/bpf: Add testcases for BPF_MUL Matan Shachnai
2024-12-30 23:00 ` [PATCH bpf-next v4 0/2] bpf, verifier: Improve precision of BPF_MUL patchwork-bot+netdevbpf
2 siblings, 0 replies; 4+ messages in thread
From: Matan Shachnai @ 2024-12-18 3:23 UTC (permalink / raw)
To: ast
Cc: harishankar.vishwanathan, srinivas.narayana, santosh.nagarakatte,
    m.shachnai, Matan Shachnai, Daniel Borkmann, John Fastabend,
    Andrii Nakryiko, Martin KaFai Lau, Eduard Zingerman, Song Liu,
    Yonghong Song, KP Singh, Stanislav Fomichev, Hao Luo, Jiri Olsa,
    Mykola Lysenko, Shuah Khan, Cupertino Miranda, Menglong Dong, bpf,
    linux-kernel, linux-kselftest
```

This patch improves (or maintains) the precision of register value tracking in BPF_MUL across all possible inputs. It also simplifies scalar32_min_max_mul() and scalar_min_max_mul().

As it stands, BPF_MUL is composed of three functions:

```
case BPF_MUL:
    tnum_mul();
    scalar32_min_max_mul();
    scalar_min_max_mul();
```

The current implementation of scalar_min_max_mul() restricts the u64 input ranges of dst_reg and src_reg to be within [0, U32_MAX]:

```
/* Both values are positive, so we can work with unsigned and
 * copy the result to signed (unless it exceeds S64_MAX).
 */
if (umax_val > U32_MAX || dst_reg->umax_value > U32_MAX) {
    /* Potential overflow, we know nothing */
    __mark_reg64_unbounded(dst_reg);
    return;
}
```

Upstreamed a few months ago to
bpf-next

Looking Ahead?

Specify the verifier at a higher level of abstraction?

Verifier in user space?

Automatically Checking the Precision of Operators?

A compiler explorer like framework for the eBPF verifier? Patches with correctness arguments

**“Always-on” Lightweight Formal
Methods have the potential to make
the eBPF verifier robust**

Open Source

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

