Addressing Invariant Violations and Improving Dead Branch Detection

Hari

RUTGERS

Srinivas

RUTGERS

Santosh

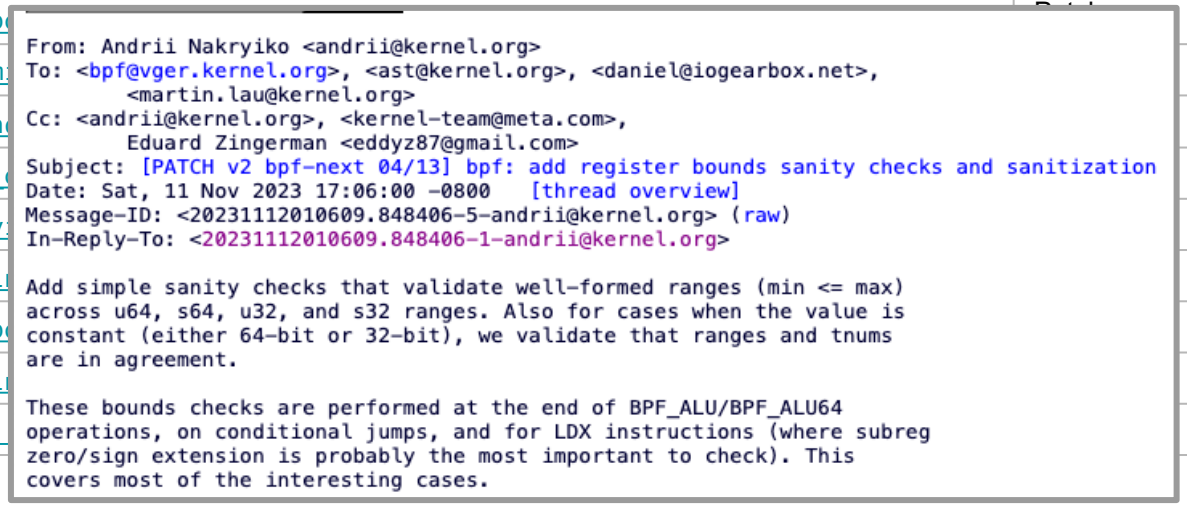
RUTGERS

Paul

ISOVALENT

A History of Invariant Violations

Description	Type	Date	Kernel
bpf: reg invariant voilation after JSET	Bug Report	21 Nov 2023	v5.19 (?)
bpf: reg invariant voilation after JSLE	Bug Report	21 Nov 2023	v5.19
bpf: Forget ranges when refining tnum after JSET	Patch	10 Jul 2025	v6.17-rc1
bpf: Improve b		19 Jul 2025	v6.17-rc1
BPF verifier m		17 Oct 2025	6.18.0-rc1
bpf: Skip boun		3 Nov 2025	v6.19-rc1
bpf: Fix tnum		27 Oct 2025	
range bounds v		27 Dec 2025	v6.19-rc2
bpf: Prevent i		20 Feb 2026	
bpf: Improve b		27 Feb 2026	
bpf: Prevent i		20 Feb 2026	
selftests/bpf:		16 Apr 2026	



```
From: Andrii Nakryiko <andrii@kernel.org>
To: <bpf@vger.kernel.org>, <ast@kernel.org>, <daniel@iogearbox.net>,
        <martin.lau@kernel.org>
Cc: <andrii@kernel.org>, <kernel-team@meta.com>,
        Eduard Zingerman <eddyz87@gmail.com>
Subject: [PATCH v2 bpf-next 04/13] bpf: add register bounds sanity checks and sanitization
Date: Sat, 11 Nov 2023 17:06:00 -0800   [thread overview]
Message-ID: <20231112010609.848406-5-andrii@kernel.org> (raw)
In-Reply-To: <20231112010609.848406-1-andrii@kernel.org>

Add simple sanity checks that validate well-formed ranges (min <= max)
across u64, s64, u32, and s32 ranges. Also for cases when the value is
constant (either 64-bit or 32-bit), we validate that ranges and tnums
are in agreement.

These bounds checks are performed at the end of BPF_ALU/BPF_ALU64
operations, on conditional jumps, and for LDX instructions (where subreg
zero/sign extension is probably the most important to check). This
covers most of the interesting cases.
```

Invariant Violations – What are they?

- Verifier's `is_branch_taken()` logic is imprecise
  - Computes whether branch will or will not be taken at runtime
- Sometimes, a branch is impossible
  - But given current register state (ranges and tnums for registers), verifier is unable to determine that branch is dead
  - Instead of pruning the branch, verifier explores it
- Along the dead branch, verifier updates register state
  - Use the branch condition to refine the ranges and tnums
  - End up with ill-formed ranges (e.g. `umin > umax`)
  - The internal invariant (all ranges should be well formed) is violated.
- Reported as: `verifier bug: REG INVARIANTS VIOLATION`
  - Register ranges and tnums are set to unbounded `[INT_MIN, INT_MAX]`
  - Continue verification

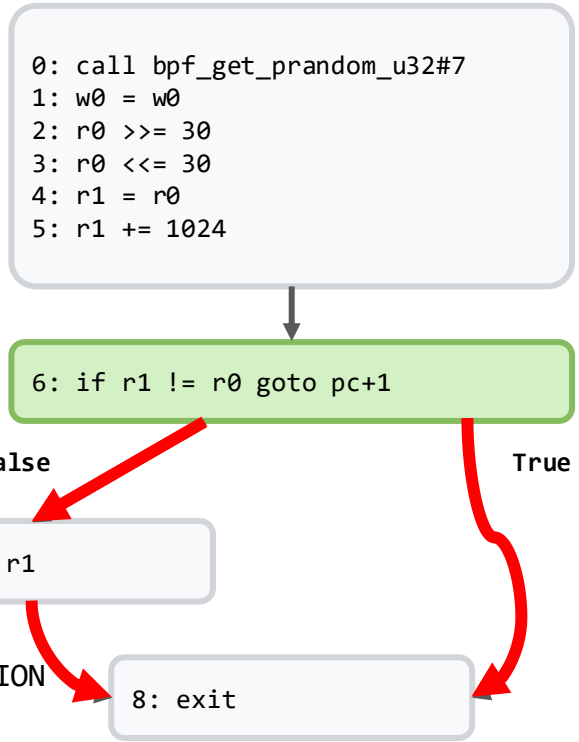
Causes And Implications

- Imprecision due to over approximation is unavoidable
  - Domain cannot represent disjunctive constraints
    - ; r2 = [1, 7]
      if r2 != 4 goto +2 ; r2 = [1, 3] ∨ [5, 7] ⟹ [1, 7]
  - Does not keep track of relationships between registers
    - ; r1 = [0, 5], r2 = [0, 5], r1 + r2 < 5
      r1 + r2 > 5 ; ⟹ always FALSE
- Implications of invariant violations?
  - At best, a warning that doesn't affect safety of programs
  - At worst, an imprecision issue that can cause programs to be rejected

Verifier Code for Jumps (roughly)

```
def check_cond_jmp_op(insn, src_reg, dst_reg):

    # determine if true/false branch is *definitely* taken
    pred = is_branch_taken(insn, src_reg, dst_reg)

    if pred == 0:
        insn_idx += insn.off  # only follow the TRUE branch
        return
    elif pred == 1:
        insn_idx += 1  # only follow the FALSE branch
        return

    # fork new verifier state for other branch
    other_branch = push_stack()
    other_src_reg = other_branch.src_reg
    other_dst_reg = other_branch.dst_reg

    # update reg states based on branch condition
    regs_refine_cond_op(insn, src_reg, dst_reg, other_src_reg, other_dst_reg)

    # sync tnum and ranges for all regs
    reg_bounds_sync()

    # check for invariant violations e.g. (umin > umax) for all regs
    reg_bounds_sanity_check()
```

An Example



u64 range:
r0 = [0, 0xc0000000]
r1 = [1024, 0xc0000400]

WARNING: REG INVARIANT VIOLATION

Fix: Use tnum information in `is_branch_taken()` for `BPF_JNE`

```
tnum:
r0 = xx00000000000000000000000000000000
r1 = xx00000000000000000001000000000000
```

https://lore.kernel.org/bpf/ba9baf9f73d51d9bce9ef13778bd39408d67db79.1755098817.git.paul.chaignon@gmail.com/

General Pattern for “Fixing” Invariant Violations

- Improve `is_branch_taken` logic
  - Use additional information available in register states to make branch taken decisions
- Improving `reg_bounds_sync()`
  - Make tnums ⟺ ranges refinement tighter, next instruction's `is_branch_taken()` has more precise information, and can prune the branch

But…Agni Already Verified Jumps?

- Agni: A formal verification tool for the eBPF verifier
  - Automatically verifies correctness of ALU and JUMP eBPF operators
  - Has been verifying since v5.17, reports JUMP are sound.
  - What gives?
- Let's look at what Agni verifies:

- On the branches taken at runtime
  - Register states will contain all possible runtime values
- If verifier follows the non-taken branch:
  - All bets are off

$P, Q \in$ **tnums × u64 × s64 × u32 × s32:**

$x, y \in$ **integers**:

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

$$P_t, Q_t, P_f, Q_f = \texttt{check\_cond\_jmp\_op}_{JNE}(\mathrm{P}, \mathrm{Q}) \quad \wedge$$

$$x \neq y \implies member(x, P_t) \wedge member(y, Q_t) \quad \wedge$$

$$x = y \implies member(x, P_f) \wedge member(y, Q_f)$$

Patch: Fix invariant violations and improve branch detection

- 🚨Idea: It must be the case that ill-formedness, i.e., invariant violations can ***only occur*** on the branches ***are never possible*** at runtime
  - If we ever have invariant violations – we must be on a dead branch.
  - Why not use invariant violations as a signal to prune branches!
- Fold in `regs_refine_cond_op()` into `is_branch_taken()'s` tail.
  - Compute new ranges and tnums for both branches, assuming both branches will be taken.
  - Don't fork verifier state yet.
- If we encounter ill-formed register states (violation) along a branch, that branch must be dead. Prune that branch.
- If no violation is encountered, both branches must indeed be possible.
  - Fork the verifier state
- Merged

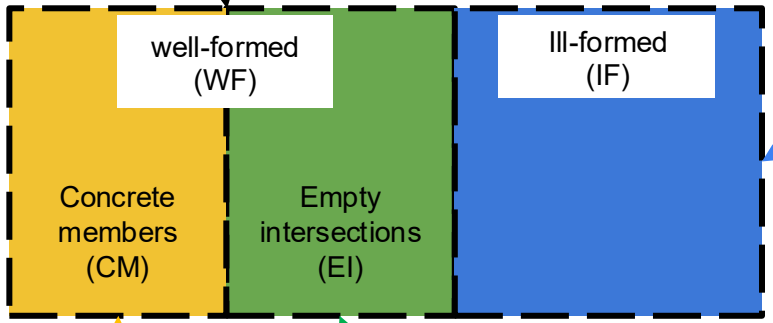
Can We Do Better?

- What is the actual invariant the verifier must maintain?
  - One invariant we already looked at: don't have ill-formed ranges and tums
  - Anything else?
- There should be at least one concrete value x such that it is contained across all abstract values in the register state:
  - $x \in \texttt{tnums} \land x \in \texttt{s64} \land x \in \texttt{u64} \land x \in \texttt{u32} \land x \in \texttt{s32}$

The Verifier State Space

All abstract values individually well-formed

```
umin <= umax ∧
smin <= smax ∧
s32min <= s32 max ∧
u32min <= u32max ∧
tval & tmask == 0
```



At least one abstract value not well-formed

```
umin > umax ∨
smin > smax ∨
s32min > s32 max ∨
u32min > u32max ∨
tval & tmask != 0
```

Aleast one common concrete member x across all abstract values

```
u64 = [1,  5], u32 = [1 , 1],
s64 = [-8, 2], s32 = [1, 1],
tnum = 000x {0, 1}, x = 1
```

No single concrete value contained across all abstract values

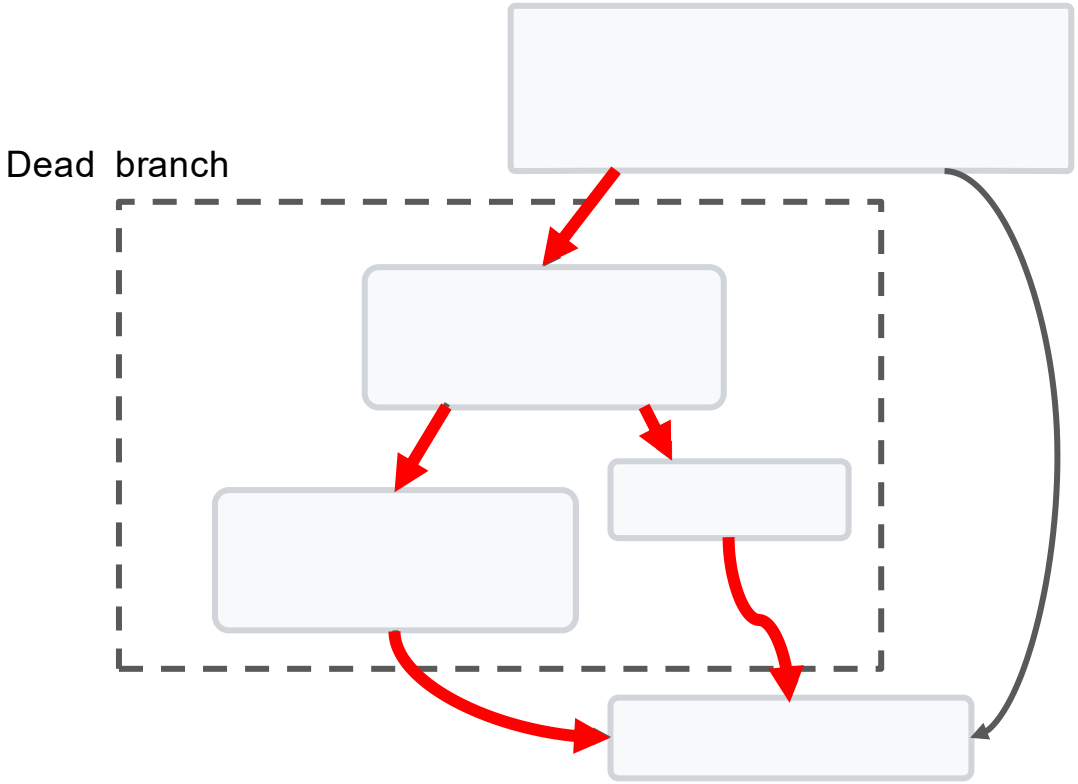
```
u64 = [3, 3], tnum = 01x0 {4, 6}
```

The Verifier State Space

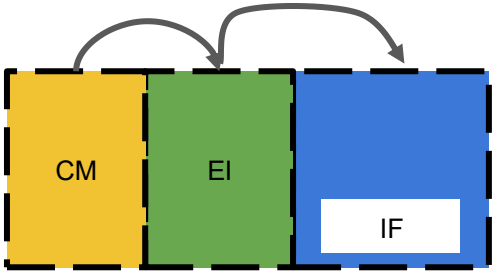
Represents non-empty sets.
Always stay in this zone.

Represents null sets.
Avoid getting here.

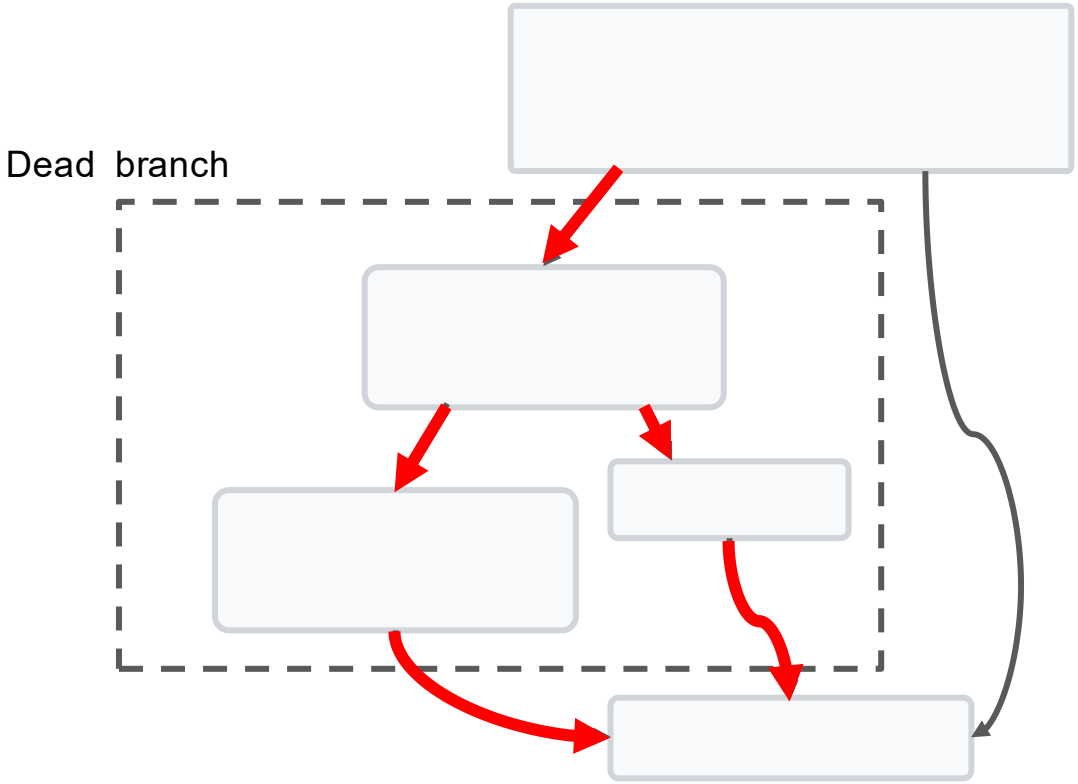
Execution Paths



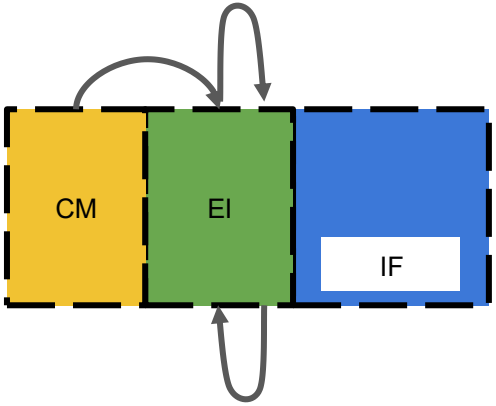
① We can go from
CM – EI – IF



Execution Paths



② We can go from CM – EI, then keep going in circles

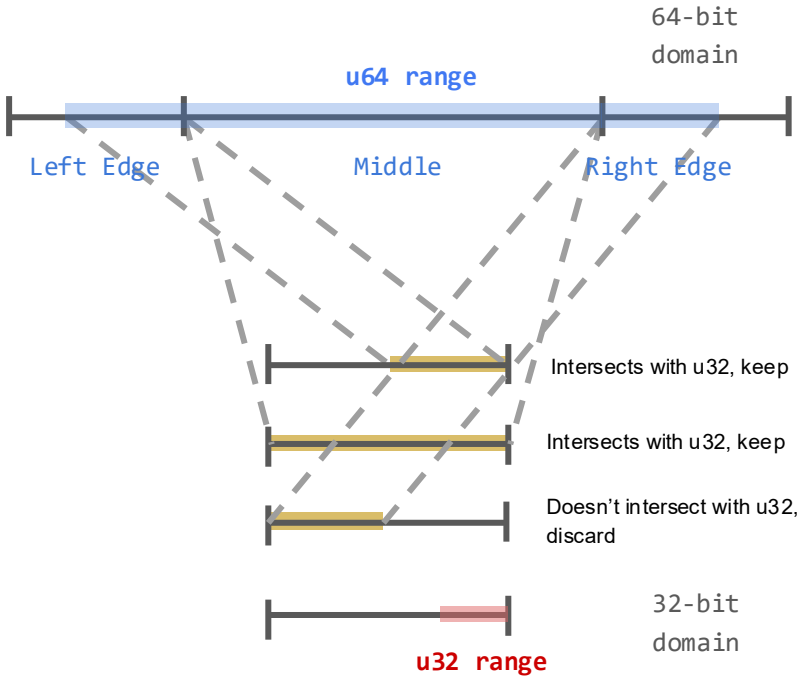


Important that we can detect empty intersections

1. Catch all cases
2. Catch them early

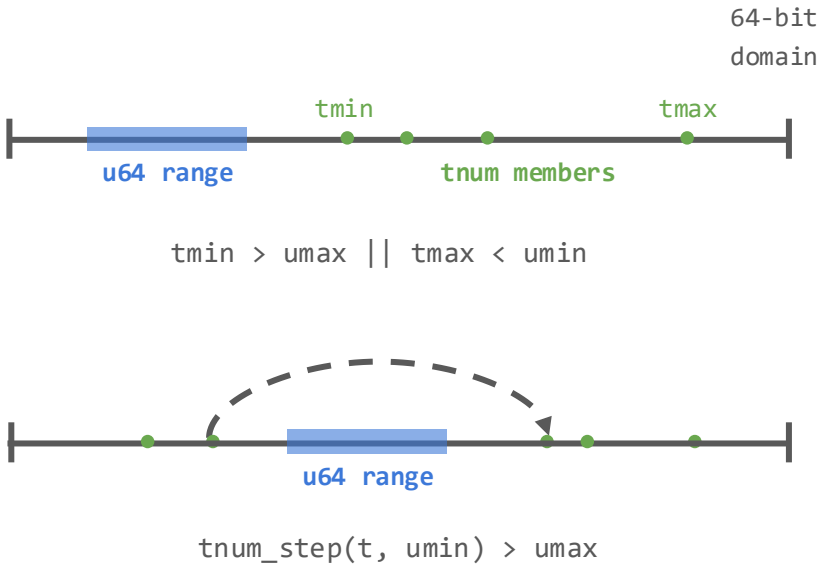
Detecting Empty Intersections (u64, u32 and tnums)

- u64 and u32
  - Intersect the ***projection*** of the u64 range onto the 32-bit domain, with the u32 range
  - The u64 range can span multiple 32-bit subspaces (at most 3); decompose the u64 range into 32-bit subspaces by projecting
  - Filter subspaces that have an intersection with the existing u32 range



Detecting Empty Intersections (u64, u32 and tnums)

- Higher order 32-bits of the collected u64 ranges may disagree with the tnum
- Further filter u64 subspaces using u64 ⟺ tnum intersection
- tnums and u64
  - Straightforward if bounding values of tnum are outside that of the range
  - What if u64 range lies entire in-between the tnum?
    - We introduced `tnum_step()`: *get the next tnum member after a given number*
    - Merged
- Sound and Complete Intersection checks!



Conclusion

- Over Approximation is inevitable
  - The verifier will explore branches that are dead at runtime
- Along those branches, if we evolve the abstract values and end up with null sets, we can detect that branch is dead
- Detecting empty intersections is a useful primitive to detect null sets when using multiple abstract domains
  - Can and should be applied to any new domains introduced into the analysis (e.g. `cnums`)
  - Domains should come equipped with operators that enable detecting empty intersections with existing domains (e.g. `tnum_step`)
- Safely prune branches once detected dead!




harishankar.vishwanathan@gmail.com