SCEV-based Loop Analysis for the BPF Verifier

LSFMM&BPF Summit 2026

Eduard Zingerman

Problem Statement

- Current verifier explores loops iteration-by-iteration (state explosion)
- Open-coded iterators require manual `bpf_loop()` / `bpf_for_each()` patterns
- Goal: automatically verify bounded `for` / `while` loops without unrolling

High-Level Approach

- Pre-analysis pass before `do_check()`
- Compute how registers evolve per iteration
- Widen register ranges at loop entry
  to cover all iterations at once
- Converge in 2 visits instead of N

```
i = 0;
while (i < 1000) {
    // i ∈ [0, 999]
    ...
    i++;
    // i ∈ [1, 1000]
}
// i = 1000
```

Pipeline Overview

Analysis passes executed before main verification pass:

1. `bpf_compute_loops()` -- loop detection: headers, backedges, latches, exits, nesting
2. `bpf_compute_idoms()` -- dominator tree
3. `bpf_compute_scev()` -- compute recurrencies for registers inside a loop

Main verification pass uses SCEV results to widen loop registers

Loop Anatomy

- **Header** -- the loop entry point, target of the backedge
- **Backedge** -- edge from inside the loop back to the header
- **Latch** -- the conditional jump that controls the backedge
  (must dominate the backedge source)
- **Exit** -- edge from inside the loop to outside

```
i = 0;
while (1) {          // <- header
    if (i >= 1000) // <- latch
        break;     // <- exit
    ...
    i++;
    continue;      // <- backedge
}
```

Dominators and latches

- Instruction A **dominates** B if every path from entry to B passes through A
- Used to find **latches**: walk up the dominator tree from the backedge source, find the nearest conditional jump with one branch exiting the loop
- "A Simple, Fast Dominance Algorithm" (Cooper et al.; ~60 SLOC)

```
i = 0;
while (1) {
    if (i >= 1000) // dominates backedge
        break;
    ...
    i++;
    continue;      // backedge source
}
```

`if (i >= 1000)` dominates `continue` ⟶ it is the **latch** for this loop

Nested Loops

- Loops form a nesting hierarchy
- Each instruction belongs to at most one innermost loop
- Inner loops analyzed first (post-order over headers)
- Outer loop treats inner loop as a black box: registers modified inside inner loop are forgotten
- Algorithm: "A New Algorithm for Identifying Loops in Decompilation"
  (Wei et al.; ~350 SLOC)

```
for (i = 0; i < N; i++) {   // outer
    ...
    for (j = 0; j < M; j++) { // inner
        ...
    }
    ...
}
```

SCEV: Goal

- **Scalar Evolution (SCEV)**: compute recurrences for registers that change inside a loop
- Express each register's value in terms of its value at the loop header
- Pattern: `reg = reg + const` $\rightarrow$ $r_k = r_0 + k \cdot \text{slope}$

```
// i = 0, p = buf
while (i < 128) {
    ...
    i++;
    p += 8;
}
```

reg	recurrence
`i`	$i_k = i_0 + k$
`p`	$p_k = p_0 + 8 \cdot k$

SCEV: Scope

- Classical SCEV builds chains of recurrences
  to describe polynomial equations: $r_k = a_0 + a_1 k + a_2 k^2 + \dots$
- For BPF we only handle **linear** recurrences: $r_k = r_0 + k \cdot \text{slope}$
- Under assumption that most real-world BPF loops are
  simple counters and pointer arithmetic

SCEV: Computing Recurrences

- Dataflow analysis over the loop body with symbolic expressions as values
- Transfer: symbolically execute each instruction
- Join: both paths agree $\rightarrow$ keep, otherwise $\rightarrow \perp$
- If every path returns to the header with the same expression for a register, that expression is the evolution formula

```
while (1) {
    if (i >= N) break;
    if (cond) {
        ...          // i = i + 1
        i++;
    } else {
        ...          // i = i + 1
        i++;
    }
                 // join: i = i + 1 ✓
}
```

Both paths agree: $i = i + 1 \rightarrow i_k = i_0 + k$

Using SCEV in the Verifier Loop

- When `do_check()` first visits a loop header, i_0 is the **concrete** register state at that point
- From latch `if (i >= N) break` + recurrence $i_k = i_0 + k$:
  $k_{\max} = (N - i_0)/\text{slope}$
- Widen register range to cover all iterations:
  $i \in [i_0,\ i_0 + k_{\max} \cdot \text{slope}]$
- Verifier explores the loop body with widened ranges
- Second visit to header: current state is a substate of the initial state $\longrightarrow$ done

```
// i_k = i_0 + k
// i_0 = 0, slope = 1, N = 1000
// => k_max = 1000
while (i < 1000) {
    // widen: i ∈ [0, 999]
    ...
    i++;
    // after i++: i ∈ [1, 1000]
    // back at header: [1,1000] ⊆ [0,1000]
    // → converged
}
// after loop: i = 1000
```

Problem: Non-SCEV Precise Registers

- Counter `i` has SCEV, pointer `p` does not
- We widen `i` to $[0, 999]$ at the header
- `p` changes each iteration with a unique value
- At the latch `i < 1000`: widened `i` allows both staying in the loop and exiting
- Each iteration explores the exit path with a distinct `p`
- Result: 1000 spurious exit states $(i{=}1000, p{=}p_1), (i{=}1000, p{=}p_2), \ldots$
- Widening made things **worse**; for programs with nested loops this can quickly lead to 1M instructions limit

```
i = 0; p = ...;
while (i < 1000) {
    // widen: i ∈ [0, 999]
    // but p has a unique value
    p = f(p);  // no SCEV
    i++;
    // latch: i ∈ [1, 1000]
    // can exit? yes (i could be 1000)
    // → explore exit with current p
}
// 1000 exit states:
// i=1000, p=p_1
// i=1000, p=p_2
// ...
```

Current Solution

- Track `iters_left` countdown per loop
- At the latch, force branch direction:
  - `iters_left > 0`: stay in loop, don't explore exit
  - `iters_left == 0`: exit, don't take backedge
- Only one exit state: $(i{=}1000, p{=}p_{final})$
- No spurious exits, no blowup
- Tradeoff: loop is still explored iteration-by-iteration ($O(N)$ states), but no multiplicative explosion

```
i = 0; p = ...;
while (i < 1000) {
    // iters_left = 999: stay in loop
    p = f(p);
    i++;
    // → forced back to header
}
// iters_left = 0: exit
// single exit state: i=1000, p=p_final
```

Current Solution: Integration with the Main Loop

```
def do_check():
  for insn in program:
    ...
    if is_next_loop_iteration():            # back at header via backedge
        maybe_clamp_scev_regs()             # tighten SCEV regs to match iteration
        iters_left -= 1
        if is_state_visited() == HIT:
            if at_terminating_loop_latch():
                push_loop_exit_state()      # push single exit state
                bpf_finalize_scev_regs()    # set post-loop register values
    if has_entered_loop():                  # first visit to header
        bpf_widen_scev_regs()               # widen SCEV registers
        loop_stack_push(iters_left=k_max)
    ...
def check_cond_jmp_op():
    is_loop_latch_taken()                   # force branch via iters_left
```

Invasive: interacts with the main verification loop logic for branch prediction, state comparison, and instruction processing.

Solutions to Explore

1. **Don't widen if some loop registers are unknown**
   - Only widen when all registers modified inside the loop have SCEV
   - Loops with non-SCEV registers fall back to iteration-by-iteration
   - No need for `iters_left` machinery
2. **Force-widen non-SCEV registers to unbounded**
   - Registers without SCEV get widened to $\top$ (full range)
   - Loop converges in $O(1)$ for all registers
   - May lose precision for non-SCEV registers

Both approaches need investigation

Current Limitations

Latch matching:

- Only 64-bit unsigned comparisons: `JLT`, `JLE`, `JGT`, `JGE`, `JNE`, `JEQ`
- Missing: signed variants (`JSLT`, `JSLE`, `JSGT`, `JSGE`), 32-bit (`JMP32`)

Dependent registers:

No SCEVs for dependent registers:

```
// Known SCEV for `i` but not for `j`
for (i = 0, j; i < 1000; i++) {
  j = i;
  ...
}
```

Register spills:

- Spilled registers are naively assumed to be non-aliased
- An additional alias analysis is needed to validate this assumption

Preliminary Test Results: Dataset

Test suites (veristat, comparing master vs SCEV branch)

Suite	Total programs	Complex (>10K verified insns)
selftests	4,656	47
scx	375	24
meta	1,540	269
cilium	134	42
Total	**6,705**	**382**

Preliminary Test Results: Wins

Among complex programs (>10K verified insns):

Category	Count
>50% insn reduction	89
10-50% insn reduction	54
Neutral (<10% change)	163

- Best: 230K-340K → 10K-41K (87-96% reduction)

Results: Regressions to Investigate

Verdict regressions (success → failure): **38 programs**

- Timeout (hit 1M limit): **5** -- scx_lavd `lavd_dispatch` (3 versions), and a few meta programs.
- Early failure: **33** -- `arena_alloc_mask` (x12), `p2dq_init_task` (x4), strobemeta `on_event` (x3), and others

Instruction count regressions (still success): **33 programs**

- a set of 29 likely similar progs: ~63K → ~80K (+27%)
- scx_lavd `lavd_init` (4 versions): 12K-27K → 144K-166K (+500-1100%)

Work in Progress

- Better main verification loop integration: avoid widening
  when some loop registers lack SCEV
- Signed (`JSLT` , `JSLE` , `JSGT` , `JSGE`) and 32-bit (`JMP32`)
  comparisons in latches
- Stack alias analysis to detect aliased spill slots
- Integration with Alexei's work modernizing the verifier as a part of a range analysis pass used to optimize out runtime checks.

Thanks!