

BPF Kernel Verifier meets GCC

A saga of arranged marriage

Vineet Gupta — Meta

eBPF track | Linux Plumbers Conference | Prague | 5 October 2026

The thesis

- ▶ The verifier is a **second back end** — it rejects programs instead of running them slowly
- ▶ GCC was never tuned for it — and it has seen **little but clang's output**
- ▶ Almost nothing here is a miscompile — it is **correct code the verifier cannot prove**

“The compiler is right, the verifier is right, and the program doesn't load.”

This has been tried before

- ▶ *The challenge of compiling for verified architectures* — Jose Marchesi, GNU Cauldron 2023 | lwn.net/Articles/946254
- ▶ *LLVM improvements for BPF verification* — Alan Jowett, LSF/MM/BPF 2024 | lwn.net/Articles/974945

Two directions explored so far:

- ▶ **Tune the optimizer** — disable or steer the passes whose output the verifier cannot follow
- ▶ **Verify during compilation** — run the verifier as a library, or model it inside the compiler

This one: start from the failures

bin every failure into categories | dig into the asm, with AI tools doing the analysis | reduce each one to a ten-line test

Act 1: The horoscopes don't match

Tests that failed; and some passed despite never running

The scoreboard

test_progs - stock kernels, no local patches, all 2026

bpf-gcc

Jan 6.19-rc1 545/4839 PASSED, 135 FAILED

Jun 7.1 599/5421 PASSED, 111 FAILED <- dissected here

Sep 7.3-rc1 659/5645 PASSED, 108/366 FAILED <- A/B base, Act 4

clang

Jan 6.19-rc1 662/5796 PASSED, 20 FAILED <- mostly env

- ▶ Between rows the **kernel and the selftests** both move — a major selftest revamp in v7.1 is much of why 135 became 111 — plus a few compiler fixes. A trend, not a controlled experiment
- ▶ The clang line is the controlled one: **same tree, only the compiler differs**
- ▶ Subtract iptables/netns/ping and clang's compiler-attributable count is **near zero**
- ▶ GCC's own testsuite barely sees BPF: tens of thousands of tests for a primary back end, ~**600** for this one

...and the green is optimistic

```
#if ... && __clang_major__ >= 18
    /* 80 real tests */
#else
    __success int dummy_test(void) { return 0; } /* reports OK */
#endif
```

- ▶ bpf-gcc doesn't define `__clang_major__`; **undefined identifier == 0 in #if**
- ▶ ~163 programs across 13 files — `verifier_sdiv.c` 80 vs. 1, `ldsx` 20 vs. 1
- ▶ So part of that PASSED column is a placeholder: **the real gap is wider than the 111**
- ▶ Fixed since — **in bpf-next**, per-feature macros in the gates and a `__skip()` tag that **actually skips them**:
 - ▶ *selftests/bpf: Enable some of the blocked cpub4 instruction tests for bpf-gcc*
 - ▶ *selftests/bpf: Report placeholder tests as SKIP, not OK*

So what are the failures?

Jun 2026 baseline: 111 top-level failures, 141 failed-load blocks

category	~	act
expected- message mismatch (negative tests)	225	Act 6
load fail -EINVAL — codegen the verifier rejects	200	Act 2
load fail -EACCES — safety rejection	42	Act 2
CO-RE / runtime mismatch	~25	—
verif_scale blowups	~12	Act 2
environment, not the compiler	~30	excluded

The **biggest** bin is message text, not broken codegen — rejected as intended, just worded differently.

The -EINVAL clusters, by size

#	verifier says	root cause
29	BPF_EXIT ... would lead to reference leak	loop rotation skips bpf_cpumask_release()
27	Caller passes invalid args into func#1	global-subprog BTF arg type -> UNKNOWN
19	At program exit R0 ... should have been in [0,1]	sign-extended return value -> Act 4
7+4	min/max value is outside ... memory range	bounds check scheduled after the add
5	Rn invalid mem access 'scalar'	arena pointer materialised as a scalar

...plus 10 stack depth 512, 9 invalid size of register fill, 6 BPF_LD_ABS

Then the kernel gained an instruction

callx — BPF_JUMP|BPF_CALL|BPF_X, Sept 2026: verifier + x86-64/arm64 JIT

- ▶ The encoding is ancient. Alexei, **2019**: *“llvm supports indirect calls since 2017 ;)”*
— only the verifier said no; bpf_tail_call() was the workaround for seven years
- ▶ Day one, the bpf-gcc selftest build **broke**
- ▶ GCC emits callx only under -mxbpf, and its **pseudo-C output didn't assemble**:
Error: unexpected register name 'r0' in instruction 'call r0'
- ▶ No __BPF_FEATURE_CALLX anywhere -> selftests can only say #ifdef
__clang__

Act 2: The first real arguments

Same C , different outcomes — four ways to be right and still be rejected

Case 1: EVRP deletes the only bound

bpf_iter/unix — dump_unix(), printing an abstract socket's name

```
for (i = 1; i < len; i++) {  
    if (i >= sizeof(struct sockaddr_un))    /* constant 110 */  
        break;  
    ... sun_path[i] ...  
}
```

clang	bpf-gcc
if r1 >= r8 goto .. ; i < len	r7 += -0x2 ; trip = len-2
if r9 < 0x6d goto .. ; i < 109	if r5 != r7 goto -56 ; i != trip
^ CONSTANT	no 0x6d / 0x6e anywhere

- ▶ -faggressive-loop-optimizations proves $i > 109$ unreachable from the `sun_path[108]` access — and **deletes the break**
- ▶ Verifier walks i to $\sim 8181 \rightarrow$ **“The sequence of 8193 jumps is too complex”**

Case 1: the A/B

build	verdict	insns / states
-O2 (default)	failure	116,676 / 1157
-fno-aggressive-loop-optimizations	success	6,596 / 143

- ▶ Same source, same compiler, **one flag** — 18x fewer insns, 8x fewer states
- ▶ GCC may treat undefined behaviour as the **programmer's** problem. The verifier has no such luxury: it has to reject what it cannot prove

*The verifier bounds loops on < **against constants**. != **against an unbounded runtime value gives it nothing**.*

Case 2: turning unrolling off is harder than it looks

arena_htab_asm explodes: **1M insns / 80,008 states** (verifier insns processed / states explored). Unrolling is the cause — so switch it off:

- ▶ `-fno-unroll-loops` -> **no change**, still 1M
- ▶ `-funroll-loops` unrolls loops with a **known** trip count; `-funroll-all-loops` unrolls **every** loop
- ▶ BPF enables the **stronger** one by default at every -O level, and the pass runs if **either** is set

```
bpf-common.cc: { OPT_LEVELS_ALL, OPT_funroll_all_loops, NULL, 1 },  
loop-init.cc : flag_unroll_loops || flag_unroll_all_loops || ...
```

- ▶ `-fno-unroll-all-loops` -> **148 insns / 12 states** — explosion gone (*still rejected, for an unrelated `addr_space_cast` bug*)

...so make it the default? It regresses four tests

test	exit test	rejection
map_lock	if r0 != r2, map_value(vs=68)	off=68 size=4
may_goto_2	if r0 != r10, R0=fp0	invalid read from stack
xdp_dynptr	if r8 != r1, mem(sz=48)	off=48 size=2
verif_scale3	if r3 != r9, pkt(r=121)	off=121 size=4

- ▶ Without unrolling GCC emits a **pointer-walking loop**, if `ptr != end` — and **the verifier does not fold pointer-vs-pointer comparisons**
- ▶ So it explores the continue edge with the pointer one step past the end
- ▶ Signature: the rejected offset **is** the region size (68, 48, 121) — the access starts at the first byte past the end, never deeper

Case 3: the check is there, in the wrong place

task_local_data — task_main(), indexing a per-task array
same shape in fs_kfuncs, sock_fields/unix_listen, tcp_custom_syncookie

```
52: (89) r2 = *(s16 *)(r7 +4)      ; smin=-32768 smax=32767
53: (d5) if r2 s<= 0x0 goto pc+78 ; lower bound -> R2=[1,32767]
54: (0f) r4 += r2                  ; <-- the add happens HERE
55: (25) if r2 > 0xfdf goto pc+72 ; <-- upper bound only AFTER the add
56: (79) r0 = *(u64 *)(r4 +0)
R4 max value is outside of the allowed memory range
```

- ▶ The C source checks both bounds; GCC scheduled the add between them

Instruction scheduling is a verifier-visible decision. Nobody told the scheduler.

Case 4: GCC likes sign extensions

BPF's PROMOTE_MODE currently widens every sub-64-bit integer to **DImode**, so a narrow signed value gets sign-extended on its way into a register:

```
/* gcc/config/bpf/bpf.h */
if (GET_MODE_CLASS (M) == MODE_INT && GET_MODE_SIZE (M) < 8)
    M = DImode;

int f (void)                f:
{
    int r = foo ();          call    foo
    if (r > 5)                r1 = (s32) r0    <-- here
        return 0;            if w0 s<= 5 goto .L2
    return r;                 r1 = 0
                                r0 = r1
                                exit
}
```

- ▶ An int return, a narrow signed argument used as an index, an int read from memory and scaled — each picks up a (s32)
- ▶ clang mostly does not. **That difference is the next two acts**

Act 3: First one to change

So fix GCC — what landed, what didn't, and what was never going to

Despite being right

- ▶ GCC promotes narrow arguments in the **callee**. LLVM does it in the **caller**. Same C, two calling conventions
- ▶ LLVM's behaviour inherits a **2012 x86-LLVM bug** (llvm#12579) that violates the x86 psABI — so by the letter, GCC is the correct one
- ▶ Years of BPF programs are built on it. Hyrum's law wins
- ▶ Verdict from the BPF community: **fix GCC** (PR124171)

Being right is not the same as being listened to. In this marriage, compatibility outranks correctness — and the fix queue is ordered by whose failures are cheaper.

The one that moved the needle

A cost-model question: reuse a register that already holds the value, or load the constant again? GCC priced the reuse cheaper.

```
ret = bpf_copy_from_user_str(sizeof(buf));    /* buf[4] */  
if (ret != 4) return false;  
ret = bpf_copy_from_user_str(sizeof(buf));    /* so ret is 4 here */
```

before	after
r1 = 4	r1 = 4
call bpf_copy_from_user_str	call bpf_copy_from_user_str
r1 = (s32) r0 <- reuse r0	...
...	r1 = 4 <- reload the constant
call bpf_copy_from_user_str	call bpf_copy_from_user_str

- ▶ Reusing r0 looks free — control flow already proves it is 4...
- ▶ ...except it drags a (s32) with it, and the verifier loses the bound
- ▶ TARGET_RTX_COSTS: a 32-bit CONST_INT is **free** — e65d38a0dc84

-Summary: 596/5387 PASSED, 123 SKIPPED, 114 FAILED

+Summary: 600/5422 PASSED, 123 SKIPPED, 110 FAILED

Three fixes, three outcomes

the fix	what happened
zero_extendsidi2 emitted <code>r0 = r0</code> , not <code>w0 = w0</code> — upper 32 bits never cleared (PR124670)	landed upstream bd275e81812c — for selftests, a wash
TARGET_RTX_COSTS: a 32-bit <code>CONST_INT</code> is cheap, so stop reusing a live register for a constant argument	landed upstream e65d38a0dc84 — +4 tests
PROMOTE_MODE: promote to SI mode, not DI mode — stop widening narrow values to 64 bits (PR124419)	more <code>wN</code> regs, fewer <code>(s32)</code> — and +1 / -6 tests

A real bug, fixed correctly, changed nothing. A cost-model tweak fixed four tests. And going after the root cause turned out to need more work than the problem it was solving.

Act 4: The verifier meets GCC halfway

GCC as the stress test the verifier never had

Linked scalars, and where the link breaks

The verifier tracks **equal scalars**: $rX = rY$ hands both the same id, so a branch that narrows one narrows the other.

```
call bpf_get_prandom_u32      ; r0 unknown
r1 = r0                       ; linked: r0, r1 share a scalar id
if r1 > 100 goto L            ; narrows r1 ...
                               ; ... and r0 with it (sync_linked_regs)
```

Put **one sign extension** in the middle (Act 2, Case 4) and the link is gone —
bpf_sockmap_map_iter_fd, program copy:

```
32: r1 = r0                    ; R0=scalar(id=4) R1=scalar(id=4)  <- linked
33: r0 = (s32)r0                ; R0=scalar(smin=0xffffffff80000000,smax=0x7fffffff)
34: if w1 == 0x0 goto 41        ; taken: w1 == 0, so r0 == 0 -- provably
41: exit
```

At program exit the register R0 has smin=-2147483648 smax=2147483647
should have been in [0, 1]

The fix: split the relation in two

An id can only say **these registers hold the same value, give or take a constant**.
`r0 = sext32(r1)` is not that — but it factors into two relations the verifier already trusts:

- (a) `r0.lo == r1.lo` ; plain equality, scoped to 32 bits
- (b) `r0.hi = fill(bit 31 of r0.lo)` ; r0's own halves -- no second register

- ▶ **(a) a subreg link kind** — delta-0, but copies *only* the 32-bit fields; a plain link today does a full 64-bit `copy_register_state`
- ▶ **(b) sext_width on the register** — high half recomputed in `reg_bounds_sync` every time the low half narrows

Replay insns 33–41: `w1 == 0 -> w0 == 0 -> r0.hi = sext(0) = 0 -> r0 == 0`,
inside `[0, 1]`, accepted

v2, currently being discussed on [bpf@vger](https://lore.kernel.org/bpf/20260910164635.459558-1-vineet.gupta@linux.dev) |
lore.kernel.org/bpf/20260910164635.459558-1-vineet.gupta@linux.dev

What it bought

Same kernel baseline / same gcc / only the verifier series applied

	Sep baseline	+ series
Summary	659/5645, 108/366 FAILED	665/5691, 102/345 FAILED
load-failure blocks	159	133
At program exit ... — all ranges	21	0
8193 jumps too complex	6	0

6 top-level tests / 21 subtests fixed, 0 top-level regressions

bpf_ip_check_defrag | bpf_sockmap_map_iter_fd | ksock_lsm |
setget_sockopt | sockmap_basic | verify_pkcs7_sig

...and one honest regression

iters/iter_obfuscate_counter — OK before, FAIL after

```
21: (bf) r6 = (s32)r6    ; smin=0xffffffff80000000 smax=0x7fffffff
```

BPF program is too large. Processed 1000001 insn

```
... max_states_per_insn 65  total_states 33341
```

- ▶ Register ids run to **66668**; the counter stays a precise constant every iteration -> the loop is enumerated, not pruned
- ▶ The trigger is (s32) **inside a loop body** — a fresh id every iteration, so no two states compare equal and pruning blows up
- ▶ **Still open**

Act 5: The in-laws

binutils, libbpf, pahole

A wrong turn in GCC, a fix in gas

```
/tmp/cc7214hK.s:40: Error: unexpected register name `w5' in expression
      *(u32 *) (r9+0) = w5
/tmp/ccVrSnkD.s:3697: Error: unrecognized instruction `w3 =*(u32*)(r10+-96)'
```

- ▶ Both are **valid per the ISA spec**; gas simply didn't have the patterns
- ▶ Found by a GCC experiment that was itself wrong and got thrown away — relaxing `bpf_print_register` to always emit `wN` for SI operands
- ▶ Landed in binutils anyway: 774c54454b1f, 350f9e8adbf2 (PR 34029)

The discarded experiment was the only reason anyone looked.

gas mis-assembled a documented mnemonic

- ▶ `gotol +1` matched the shorter `goto` template; the trailing `l` was parsed as the start of the offset
- ▶ Emitted **JA (0x05) instead of JAL (0x06)**, plus a relocation against an invented undefined symbol `l`
- ▶ **No diagnostic.** Only the signed forms were affected, which is why it hid
- ▶ Fixed upstream: a `%t` “end of mnemonic” tag, now in 66 templates — [92529f776514](#), [c8b5c0f50e2e](#) (PR [gas/34558](#))

Other fragile seams between projects

BTF tags — `btf_decl_tag` / `btf_type_tag` now work with GCC, but the kernel still says otherwise:

```
config PAHOLE_HAS_BTF_TAG
    depends on CC_IS_CLANG          # "only clang implements these"
```

- ▶ pahole side **landed**; the kernel patch is a three-file Kconfig change
- ▶ Blocked on one GCC fix that **missed the 16.2 backport deadline** —
`3e2885588fe0 dwarf: fix inconsistency in type chains traversal` (PR 125421)
- ▶ So: GCC 16.3, or 17. **A one-line depends on waits a release cycle**

```
libbpf: Error loading .BTF into kernel: -EINVAL. BTF is optional, ignoring.
[14] FUNC 'memcpy' type_id=1      Invalid func linkage
```

- ▶ **20 of 1005** gcc objects: an extern FUNC for a memcpy GCC then inlined away
- ▶ The program still loads — **just with no BTF**: no func/line info in verifier logs, no CO-RE against local BTF, nothing for `bpftool` to print

Act 6: Marriage counselling

Method, and what the test suite had to learn

How not to fool yourself

- ▶ Every failure mode here is **silent**: green dummies, “BTF optional, ignoring”, objects that build with zero programs
- ▶ So: **count symbols in the .o**, never trust build exit status

```
bpf-unknown-none-objdump --syms X.bpf.o | grep -cE '^[0-9a-f]+ g +F'
```

- ▶ Workflow that finds things: veristat A/B over ~4,660 bpf_gcc programs in a VM per toolchain variant -> reduce each flip to a pure-C test
- ▶ “**gcc-only symptom**” != “**gcc bug**” — the .percpu crash in that CI run was an x86 JIT bug (add_1mod: REX.B where REX.R was needed), and the moff 0 line in the defrag log is a kfunc-BTF artifact that clang produces too

The biggest bucket isn't a bug

test	sub-fails	assertion
linked_list	117	"expected error message"
spin_lock	26	"no match for expected error message"
exceptions	19	message match
verifier_subprog_precision	15	precision-log text
verifier_private_stack	4	"stack depth max ..." text

- ▶ Negative tests **rejected as intended** — with a different message
- ▶ Why they differ, one example — `iter_err_unsafe_asm_loop`: gcc loads the counter init `*(s32*)`, so `smin = -231` and the verifier stops one instruction earlier — *"unbounded min value"* instead of *"unbounded memory access"*. Both correct

"A fifth of the failure list is the test suite being surprised, not the compiler being wrong."

Asks

- ▶ **GCC folks:** focus on **feature completion** — codegen is good enough
 - ▶ CO-RE | BTF tags | arena / addr_space_cast | the remaining ISA gaps
- ▶ **Verifier folks:** bpf-gcc is a fuzzer you get for free — it emits shapes clang never will
- ▶ **Everyone:** run the bpf_gcc flavour, and don't trust a green run you haven't counted
 - ▶ None of this is in a release yet — **bpf-next, binutils master, GCC trunk**

It was an arranged marriage. They are still arguing. But the in-laws have started fixing things, and the kids — the selftests — are finally being told the truth about what they ran.

Thank You !

vineet.gupta@linux.dev

caricature: Venus Fort, Tokyo, Sep 2003



Backup B1: arena_htab_asm has two bugs

- ▶ Unrolling explains the **explosion** (1M / 80,008 -> 148 / 12)
- ▶ It does **not** explain the rejection:

```
116: r0 = addr_space_cast(r0, 1, 0)    ; R0=scalar()  
117: *(u64 *) (r6 + 0) = r0  
R6 invalid mem access 'scalar'
```

- ▶ bpf-gcc's arena `addr_space_cast` lowering loses the pointer

Backup B2: proving “hostile, not wrong”

- ▶ usdt decoupled sign-extension, both compilations loaded and run with identical inputs:

```
diff_run.stock.bpf.o      1,2,3,4,5,6,7,8,9,10,11,12
diff_run.addon.bpf.o      1,2,3,4,5,6,7,8,9,10,11,12
```

- ▶ Byte-identical output -> the delta is result-equivalent; only the verifier cares

Backup B3: the full `__clang_major__` loss

file	clang	gcc	lost
verifier_sdiv.c	80	1	79
verifier_ldsx.c	20	1	19
verifier_movsx.c	17	1	16
verifier_bswap.c	13	1	12
verifier_store_release.c	15	1	14 — gate correct
verifier_load_acquire.c	14	1	13 — gate correct

“I am using a dummy prog to avoid changing the verifier testing infrastructure. Once clang 18 is widely available ... the dummy program can be removed.” — Yonghong Song, Jul 2023 (147c8f4470ee)

Backup B4: June -> September, both stock

	Jun 2026	Sep 2026 baseline
Summary	599/5421, 111 FAILED	659/5645, 108/366 FAILED
failed-load blocks	141	159
-EINVAL / -EACCES	200 / 42	111 / 80
At program exit ... class	19 + 2	21
stack depth > 512	10	—

Backup B5: “cpu v4” is not one promise

	movsx / ldsx / sdiv / bswap	load_acq / store_rel
clang -mcpu=v4	__BPF_FEATURE_*	__BPF_FEATURE_LOAD_ACQ_STORE_REL
bpf-gcc (__BPF_CPU_VERSION__ 4)	all defined	no macro
gas -misa-spec=v4	assembles	unrecognized instruction

- ▶ v4 landed in two waves (2023 / 2025); binutils caught the first
- ▶ Gate on the **feature**, never the cpu version — ideally a capability probe