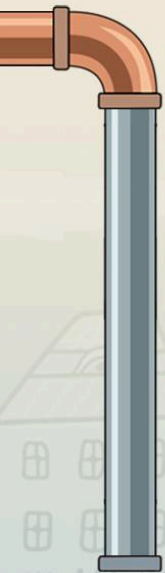


Safely Optimizing eBPF for Hardware and Workloads

kinsn · speculative ReJIT · KProg

Yusheng Zheng · Hao Sun (ETH Zurich) · Tong Yu

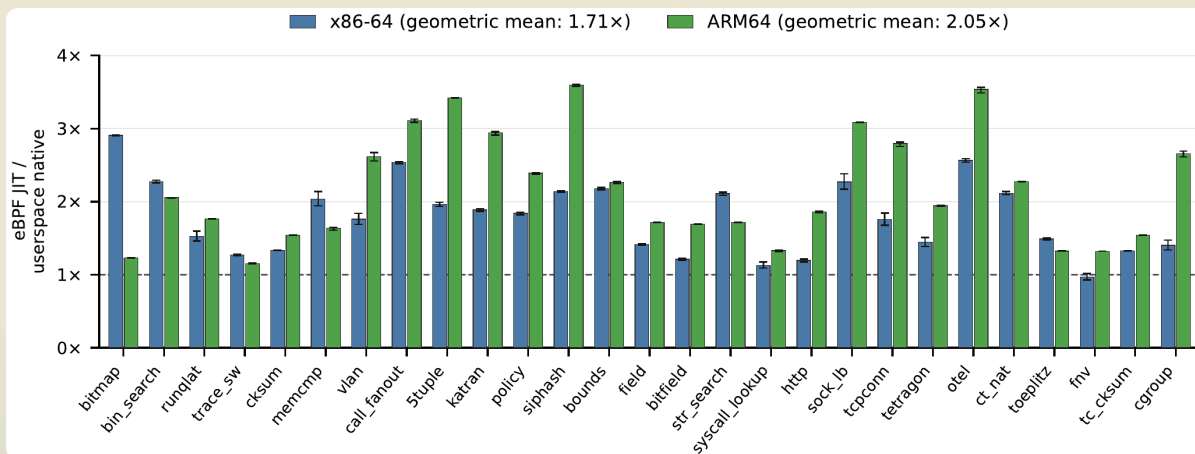
Mon 5 Oct 2026 · 17:00 · South Hall 1 A



**LINUX
PLUMBERS
CONFERENCE**

PRAGUE, CZECHIA / OCT. 5-7, 2026

eBPF compute code runs 1.7–2.1× slower than native



eBPF time ÷ native time for 27 compute functions from production eBPF programs (no helpers or maps), same C at -O2. Geomean **1.71x** x86-64, **2.05x** ARM64. Where does the time go?



The JIT is where the time goes

Two comparisons on the same 27 functions: is it the kernel, the bytecode, or the JIT?

EXECUTION-TIME RATIO, GEOMEAN OF 27	X86-64	ARM64
eBPF JIT / native, both in the kernel	1.62	1.77
Same verified bytecode through LLVM / native	1.07	1.05

Same place, 1.6–1.8× slower. Running in the kernel is not what costs the time.

Same bytecode, within 5–7%. An optimizing compiler recovers almost all of it.

The bytecode keeps the information; the gap is in code generation.



One instruction at a time: a rotate stays four

C (simplified)

```
y = (x << 16)
    | (x >> 48);
```

Clang
(native)

x86-64 native

```
rorx r15, rbx, 0x30
```

eBPF bytecode

Clang (BPF)

```
r1 = r7
r1 >>= 48
r7 <=<= 16
r7 |= r1
```

eBPF JIT

x86-64 native

```
mov rdi, r13
shr rdi, 0x30
shl r13, 0x10
or r13, rdi
```

BPF has no rotate, so LLVM spells it as copy, two shifts and an or.

The JIT translates each instruction on its own, so it emits all four.

Same for bit extract, conditional select, byte-swapped loads.

Why not teach the JIT these idioms?



JIT optimizations are trusted, per-architecture code

Trusted

The verifier checks bytecode; code the JIT emits is outside its view.

Per architecture

Each optimization is written again for x86-64, arm64, riscv, ...

Hard to get right

Jitterbug found 16 bugs in 5 deployed JITs; 7.3 fixes an x86 JIT encoding bug.

A faster eBPF has to keep new trusted kernel code small.



This talk: three approaches

kinsn · design for review

kinsn = kernel instructions supplied by modules: native instructions for common idioms that the verifier can still check.

Speculative ReJIT · research idea

Use runtime facts (hot keys, branch bias) with no kernel change. The stock verifier re-checks every rewrite.

KProg · research idea

Run whole programs natively, still judged by the stock verifier.



kinsn goal: native instructions, still verified

What we want from new native operations in the kernel.

1 Native

Use the instruction the CPU has: ROL, CMOV, BEXTR, MOVBE, LDP/STP.

2 Verified

The verifier knows exactly what the operation computes and accesses.

3 Small

One generic kernel change. New operations come from modules, not from the core.

Upstream work already covers parts of this goal.



Prior work

APPROACH	NATIVE CODE FROM	VERIFIER SEES	STATUS
Dedicated instructions discussed on bpf_ffs64, 2024	each JIT, per new opcode	the instruction	discussion
Helper and kfunc inlining in the JIT helpers 2024; Leon Hwang, bitops v2, Feb 2026	each JIT, or the compiled kfunc (proposed)	a call	helpers merged; bitops in review
kfuncs inlined as BPF Eduard Zingerman, RFC 2024; Puranjay Mohan, iterators 2026	the regular JIT	the BPF body	RFC; iterators in 7.3

kinsn builds on the last two: the verifier sees a BPF body, the JIT writes a native emit.



kinsn: a kfunc written in BPF, with a native emit

BPF body

Ordinary BPF that defines the operation. The verifier checks it at every call site.

Native emit

Machine code the JIT writes in place of the call, per architecture.

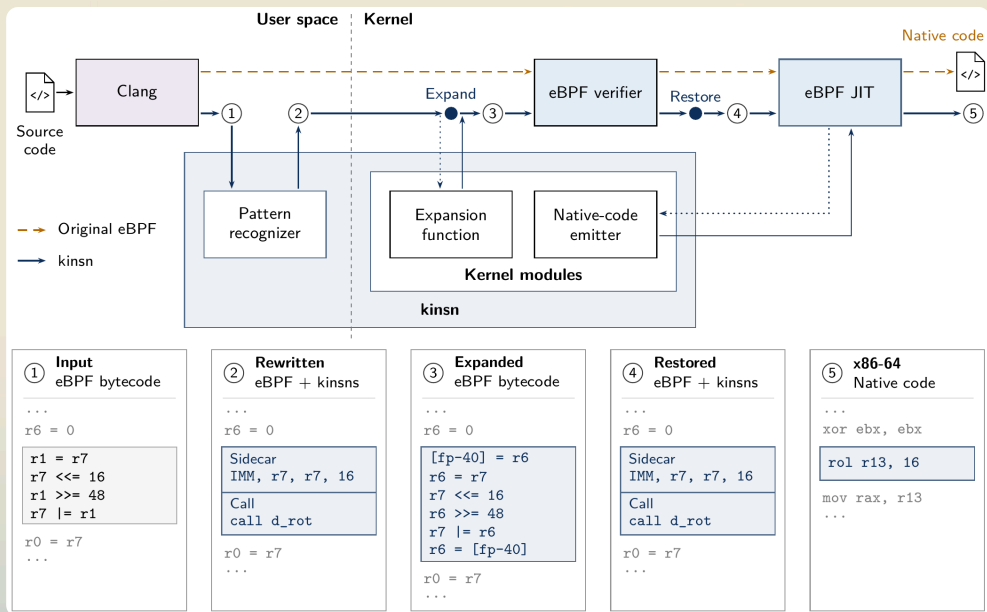
A **module** supplies both. The **compiler** requests the kfunc in place of the idiom.

OPERATION	X86-64	ARM64
Rotate	ROL	EXTR
Conditional select	CMOVcc	CSEL
Bit extract	BEXTR	UBFX
Big-endian load	MOVBE	LDR+REV
Prefetch	PREFETCHT0	PRFM
Bulk memory	MOV	LDP/STP
Address calc	LEA	—

How is one call verified and compiled?



Verify the BPF body, then emit native code



Expansion function → BPF body · native-code emitter → native emit
 In the prototype a kinsn is a **sidecar** carrying the operands, plus a kfunc call.



The native code must match the BPF body

VERIFIER GUARANTEES

The BPF body is safe at this call site, and the call passes the usual kfunc permission rules.

MODULE GUARANTEES

For every accepted operand, the native code behaves like the BPF body: same results, no other register writes or memory accesses, within its size bound.

Lean 4

proofs of equivalence for the 7 operations

100,000

random inputs per family, x86-64 and arm64

4 of 4

deliberately wrong emitters caught

New trusted code: generic kernel support plus each module's expansion and emitter.



Prototype: about 600 kernel lines, x86-64 and arm64

1	descriptors, lowering, restore	525
2	x86-64 JIT emit	45
3	arm64 JIT emit	55
4	selftests	588
5	Documentation/bpf/kinsn.rst	78

added lines, prototype against bpf-next

23 / 23

kinsn selftests, x86-64 and arm64,
jit_harden 0 and 2

0

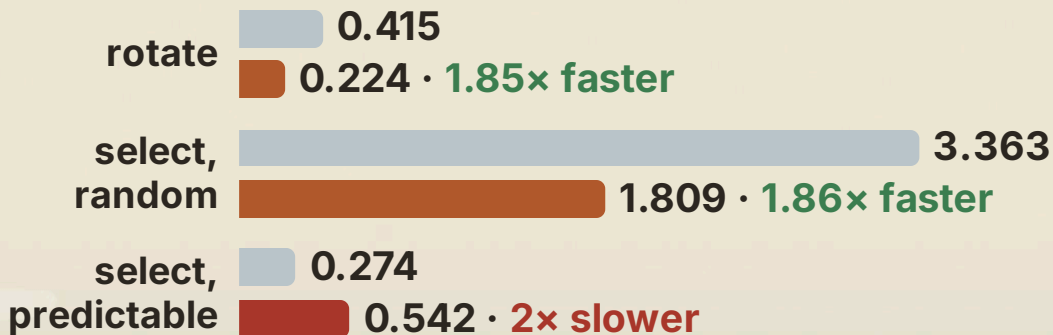
new failures in the full test_progs run

Tests cover calls, operand encoding, two kinsns in one program, mixing with kfuncs, filter rejection, saving R6 and R10, and every error path.



Wins on random branches, loses on predictable ones

ns per operation, each using the previous result: pure BPF vs kinsn (select = CMOV)

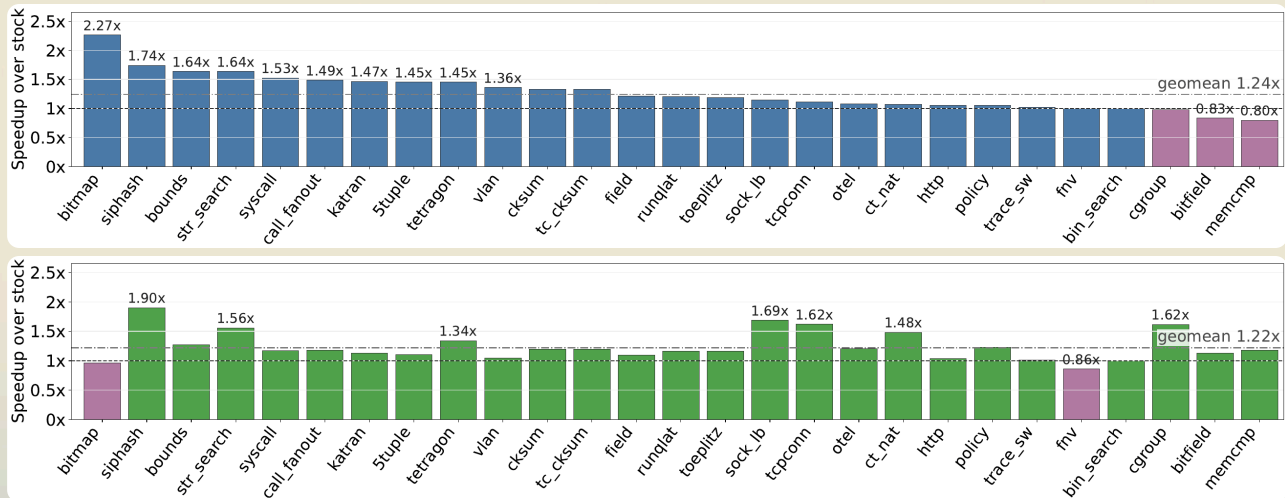


A predicted branch beats CMOV. Branch misses: 4.95 billion (BPF) vs 10.2 million (kinsn).



Seven operations: 1.24× and 1.22× on microbenchmarks

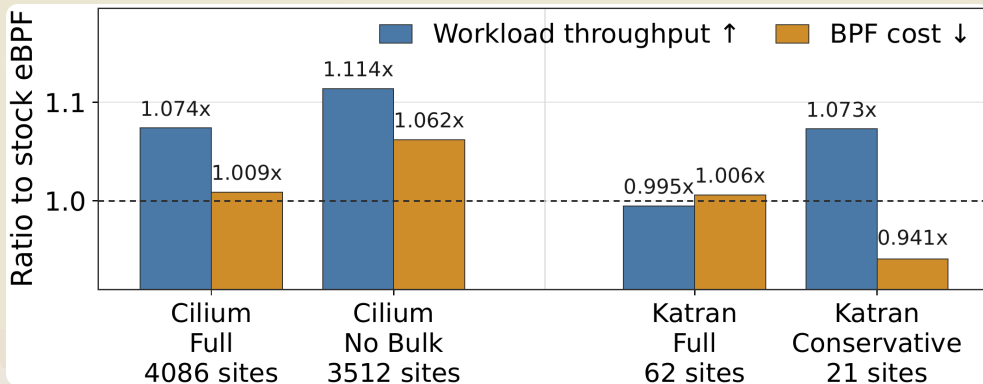
Each bar is one benchmark; purple bars are slower than the stock JIT.



Speedup over the stock JIT; top x86-64, bottom ARM64. Code size 0.77× / 0.88×.



Fewer kinsn sites can be faster



Cilium, x86-64: all 4,086 sites 1.074x;
dropping bulk copy (3,512 sites) 1.114x

Katran, ARM64: 21 chosen sites 1.073x;
all 62 sites 0.995x

Bulk copy: +28% on x86-64, about 7%
slower on ARM64

A site is one place a kinsn replaces an idiom. Which sites pay depends on the CPU and workload.



Open: encoding, native code, fallback

What the prototype does today, and what review feedback suggests for the next version.

	PROTOTYPE	OPTIONS FOR THE NEXT VERSION
Encoding	MOV sidecar, new <code>src_reg = 3</code> , + kfunc call	Plain kfunc call, operands as constant args; no new UAPI
Native code	Module emitter per architecture	Or: compiled C kfunc, copied after a disassembler check
Fallback	Refuse the load	Run the verified BPF body
Review bar	Lean 4 proofs, differential tests	What should each new operation bring?

A natural fallback: the BPF body the verifier already checked.



ReJIT goal: use runtime facts, no kernel change

kinsn decides once, at load time. "Speculative" as in JIT compilers: assume a runtime fact, guard it, fall back when it fails.

Facts kinsn cannot see

Which branch is predictable, which map entry is hot, which sites pay. Known only on the running system.

Constraint

No kernel change, unmodified applications, and the stock verifier judges every rewrite.

select: +37% on random branches, -28% on predictable ones

Katran: 21 chosen sites 1.073×, all 62 sites 0.995×

map lookups: a hot, stable key can be inlined as a constant



ReJIT specializes, the stock verifier re-checks

runner

policy: which passes, which programs, measure

No kernel change. Works on any recent kernel.

APPLICATION PROCESS (UNMODIFIED)

loader

Cilium, Katran, ...

shim

LD_PRELOAD, catches
BPF_PROG_LOAD

The optimizer is untrusted. Safety from the verifier; correctness is the pass's job.

bpftool

bytecode in, bytecode out, one pass per call

The same pipeline inserts kinsn into already-compiled bytecode. That is how the Cilium and Katran numbers were measured.

stock BPF_PROG_LOAD: verifier + JIT



ReJIT re-specializes when the workload changes

```
/* map_inline: hot key observed */  
if (key == HOT_KEY)  
    v = inlined value;  
else  
    v = bpf_map_lookup_elem(map, &key);
```

A guard miss takes the slow path in the same verified program. Frozen maps the verifier already folds itself (Linux 7.1); ReJIT targets mutable maps, hot keys, branch profiles.

On a phase change

re-profile

maps, PMU branch counts

new plan from original bytecode

stock verifier re-checks

swap in place

LINK_UPDATE, prog array, XDP, perf, tracepoint



With restart controls, only Katran gains

map_inline, throughput ratio after dividing by a restart-only control run

Katran	 1.075× control 1.00
Tetragon	 1.036×
Tracee	 1.015×
BCC	 1.008×
Cilium	 0.962× control alone 1.07

The control restarts the same application with no pass. Katran: 16 sites, 2,554 → 2,284 instructions. Measured with bpf-bench: real apps and loaders, isolated VMs.

Restarting alone moved Cilium by 7%. Without controls, that looks like a win.



KProg goal: whole programs native, still verified

kinsn replaces idioms; most datapath time is in helpers, maps and tail calls.

Whole program as one kinsn

BPF body = the program, emit = its native build. Cilium **2.358×** vs **1.074×** with kinsn idioms, but the native build is trusted.

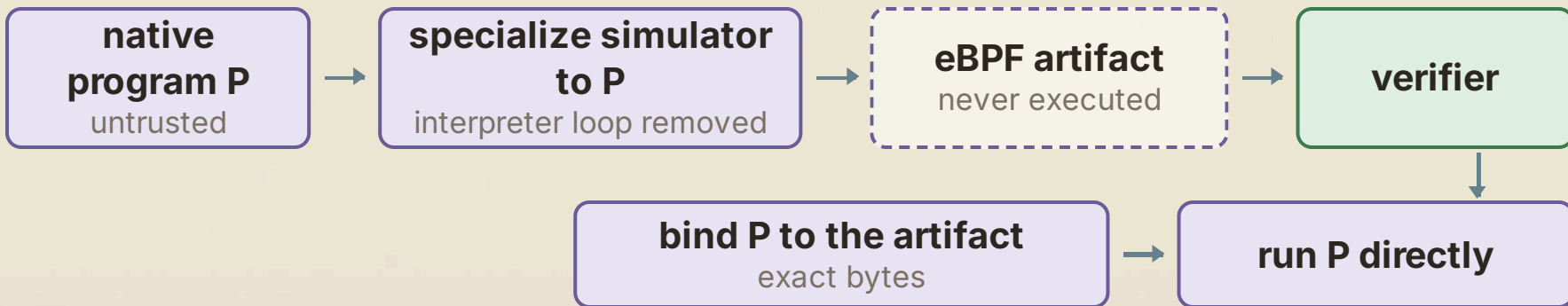
KProg: the BPF body checks the native code

Make the BPF body describe the native program itself, so the stock verifier judges what runs.

	RUNS	WHO CHECKS AN UNTRUSTED BINARY
Kernel module, Rust	native code	nobody, or the toolchain
eBPF	BPF ISA via JIT	the verifier, at load time
KProg	native code	the stock verifier, at load time



KProg: verify a simulator, run the program



Not proof-carrying code (no certificate), not SFI (no inserted checks), not translation to eBPF (no JIT cost).

Safe only if the simulator is faithful to P's memory, helper and control-flow behavior.



KProg: 1.47× headroom, fidelity still trusted

1.47×

native vs JIT, 29 microbenchmarks

Works: all 29 verify and run on x86-64 and AArch64; an out-of-bounds packet read is rejected

54×

slower to load (116 ms vs 1.9 ms median)

Still trusted: that the generator, linker and simulator match P. Lean 4 covers parts

Not yet: application results; AArch64 ran under QEMU only

A research direction. We want your view on whether this is worth pursuing.



More speed costs more trust

Where each approach sits: what it changes in the kernel, what it adds to trust, what it gains.

	KERNEL CHANGE	NEWLY TRUSTED	MEASURED
Stock eBPF	–	–	baseline
Speculative ReJIT	none	nothing: the verifier re-checks	Katran +7.5% (map_inline, x86-64)
kinsn	~600 lines (prototype)	one native emit per op and arch	1.24× / 1.22× micro; Cilium 1.074× (x86-64), Katran 1.073× (ARM64)
KProg	prototype	simulator fidelity, binding	1.47× micro, no application yet

Whole native program, trusted: Cilium 2.358×. Application numbers measured with restart controls on real loaders.



Questions for the room

kinsn design

- Is "a kfunc written in BPF, with a native emit" acceptable?
- Plain kfunc call instead of a sidecar?
- Native code from modules, or from the compiled kfunc?
- BPF body as the fallback?
- Review bar for each new operation?

Research ideas

- ReJIT: useful to you, and should "when to apply" stay in userspace?
- KProg: worth pursuing?

Paper arxiv.org/abs/2606.24213 · Code github.com/eunomia-bpf/bpf-benchmark



LINUX
PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026

