

Safely Optimizing eBPF for Hardware and Workloads

kinsn · kprog · speculative ReJIT

Yusheng Zheng · Hao Sun · Tong Yu

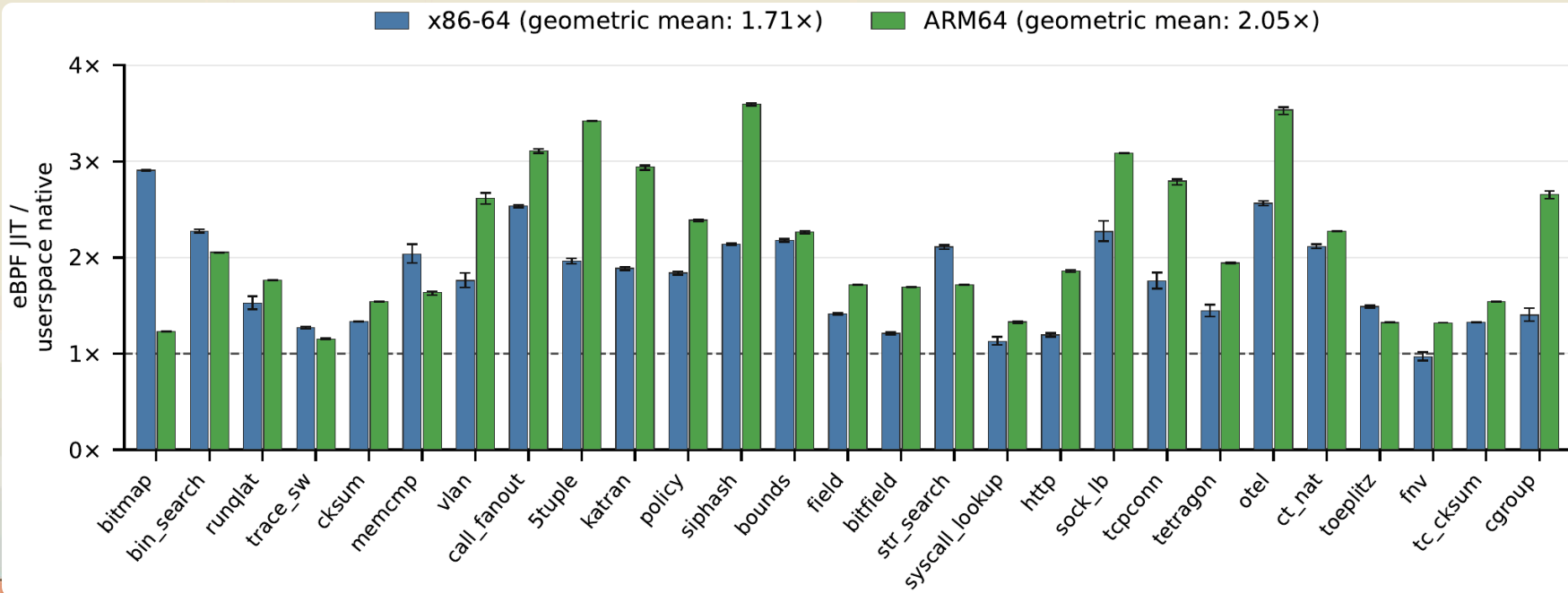
LPC 2026 · eBPF Track



LINUX
PLUMBERS
CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026

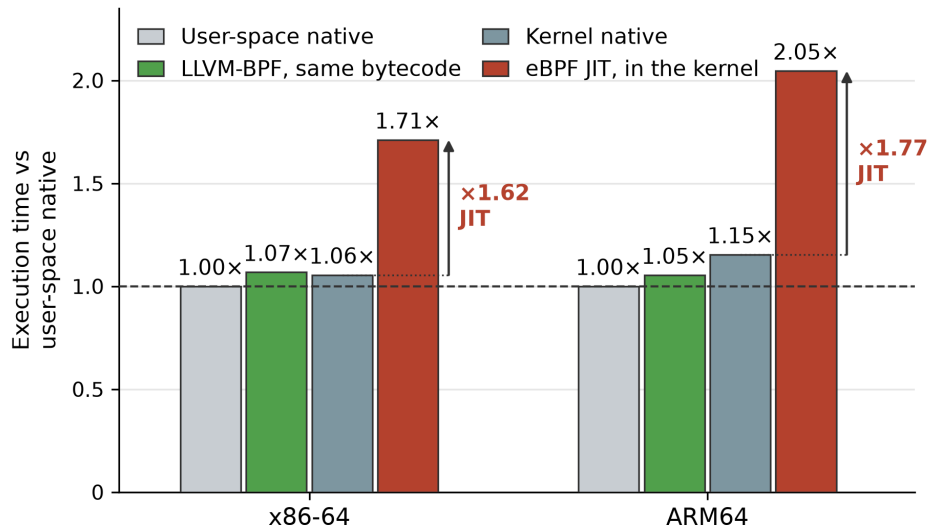
eBPF compute code runs 1.7–2.1× slower than native



The C code and the -O2 flags are the same; only the eBPF pipeline differs.



The JIT is where the time goes



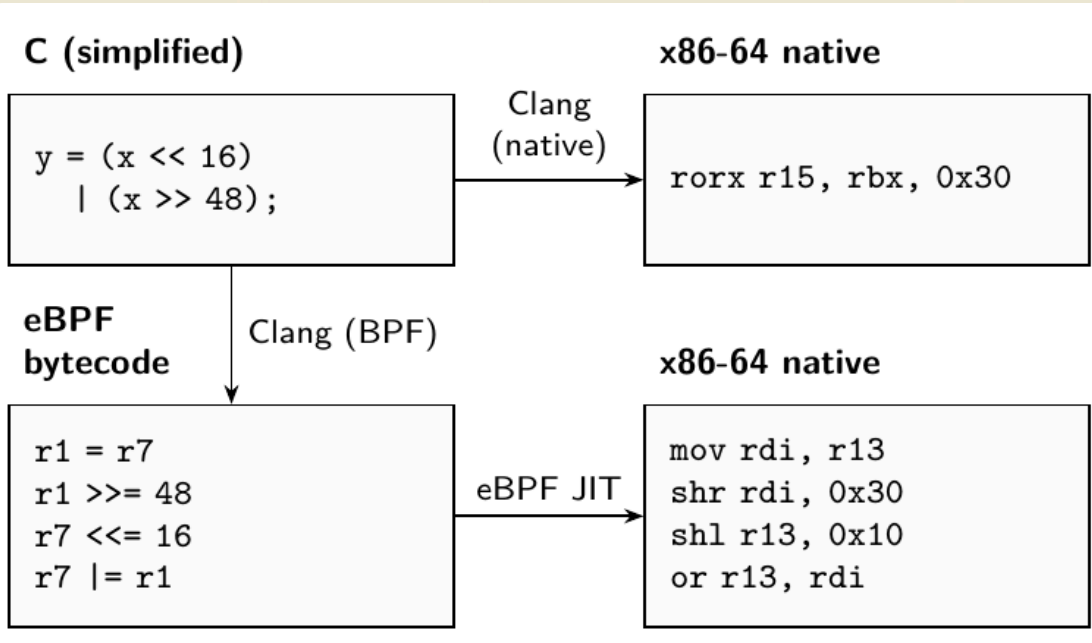
Finding 1: not the location. Native code in the kernel is only 6–15% slower. JIT code is another 1.62 $\times$ (x86-64) and 1.77 $\times$ (arm64) slower.

Finding 2: not the bytecode. LLVM compiles the same bytecode to within 7% and 5% of native.

Finding 3: the translation. The JIT translates one instruction at a time, so common operations take several instructions.



A rotate: one native instruction, four in eBPF



BPF has no rotate, so LLVM spells it as copy, two shifts and an or. The JIT translates each one on its own.

Userspace optimizers such as K2, Merlin and EPSO rewrite within the BPF ISA. They cannot emit an instruction the BPF ISA lacks.



eBPF misses both the hardware and the workload

Hardware that BPF code cannot use

The CPU has instructions that BPF cannot express.

The CPU has more registers than BPF's eleven.

Values are saved to the stack around helper calls.

Workload facts the JIT never sees

Branch predictability decides whether insn helps.

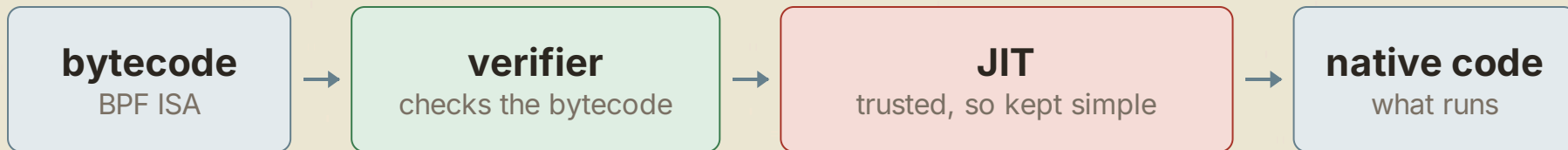
A map entry could stay the same for minutes/hours.

Which rewrites pay off depends on the traffic.

Hardware is known at load time; workload only at run time.



The ISA and the JIT are designed for safety, not speed



The ISA is made for the verifier

BPF was designed to be verifiable and portable, not fast.

The missing instructions and the few registers follow from that.

The JIT is trusted, so it stays simple

The verifier checks only the bytecode, so the kernel must trust the JIT.

An optimizing JIT would add a large trusted compiler for each architecture.

The question for this talk: how to get speed without growing TCB (trust code)?



Agenda: three approaches, then discussion

Part 1 · kinsn: verified inline kfuncs · RFC (paper in ATC'26)

Common operations get native instructions. The verifier still checks what they compute.

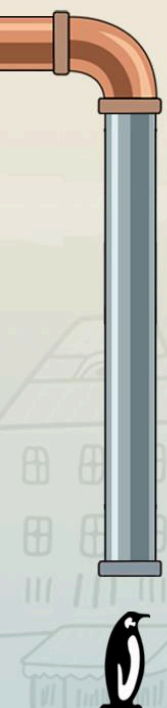
Part 2 · kprog · WIP research

A whole program runs natively. The existing verifier still decides whether it is safe.

Part 3 · Speculative ReJIT · WIP

Programs are rewritten for runtime facts, with or without kinsn and kprog. The verifier checks every rewrite.





PART 1

kinsn: verified inline kfuncs

Native instructions for common operations, checked by the verifier



LINUX
PLUMBERS
CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026

Goal: let eBPF use hardware instructions, safely

1 Fast

Common operations such as rotate run as single CPU instructions.

2 Safe

The verifier's safety result holds for the code that runs.

3 Small

Little new trusted code goes into the kernel.

4 Extensible

New operations reuse one mechanism on every architecture.

This version adds no encoding or UAPI. Native code is hand-written or copied from the compiled kfunc.

Optimizing across the whole program is not a goal; Part 2 covers it.



LINUX

PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



Why not just a kfunc? The verifier cannot see inside it

```
if (data + 12 > data_end) return 0;  
off = bpf_select64(c, 4, 8);  
v = *(u32 *)(data + off);
```

With a plain kfunc, the verifier knows nothing about the result. The load needs another bounds check.

If the kfunc also has an eBPF version, the verifier knows off is 4 or 8.

The call itself, ns per operation (x86-64)

	PURE BPF	KFUNC CALL	INLINED
rotate	0.415	0.299	0.224
select, random condition	3.363	1.875	1.809

In a chain of rotates, the call alone makes it 1.33× slower.

A kfunc hides what it computes, and every use pays for a call.



Earlier work: a new instruction, or the verifier sees a call

APPROACH	NATIVE	VERIFIER SEES	COST	STATUS
Just add instructions bit ops suggested 2024; cpubv4 in 6.6	yes	the instruction	ISA, LLVM, JITs	harder
Inlining in the JIT helpers since 6.10; Leon Hwang's RFC, 2026	yes	a call	code per JIT	helpers merged; bitops not
Kfuncs inlined as eBPF Eduard Zingerman, 2024; Puranjay Mohan, 2026	no	a call; eBPF added after	eBPF per kfunc	RFC; iterators in 7.3-rc1
kinsn (this talk) verified inline kfuncs	yes	the eBPF version	emit or copy per JIT	prototype

kinsn gets native code the verifier sees, without a new opcode.

LINUX

PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



Verified inline kfuncs: a kfunc plus an eBPF version

```
/* kernel/bpf/insn_kfuncs/bpf_insn_kfuncs.c */
__bpf_kfunc u64 bpf_rol64(u64 x, u32 n__k)
{
    return rol64(x, n__k);
}
/* the body: what the verifier checks at each call */
static const struct bpf_insn rol64_body[] = {
    BPF_ALU64_IMM(BPF_AND, BPF_REG_2, 63),
    BPF_MOV64_REG(BPF_REG_0, BPF_REG_1),
    BPF_ALU64_REG(BPF_LSH, BPF_REG_0, BPF_REG_2),
    BPF_ALU64_IMM(BPF_NEG, BPF_REG_2, 0),
    BPF_ALU64_IMM(BPF_AND, BPF_REG_2, 63),
    BPF_ALU64_REG(BPF_RSH, BPF_REG_1, BPF_REG_2),
    BPF_ALU64_REG(BPF_OR, BPF_REG_0, BPF_REG_1),
};
static const struct bpf_kfunc_body
bodies[] = {
    BODY(0, rol64, rol64_emit), ...
}; /* kfunc set: .bodies, .body_cnt */
```

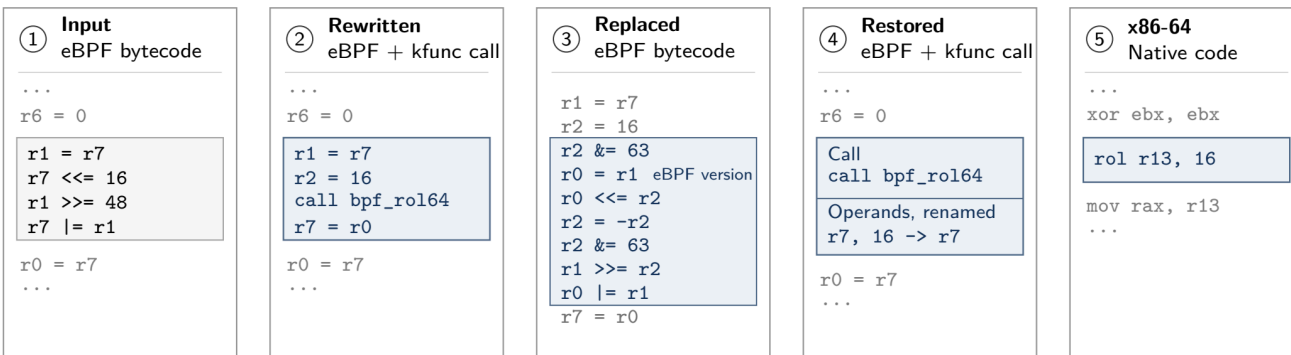
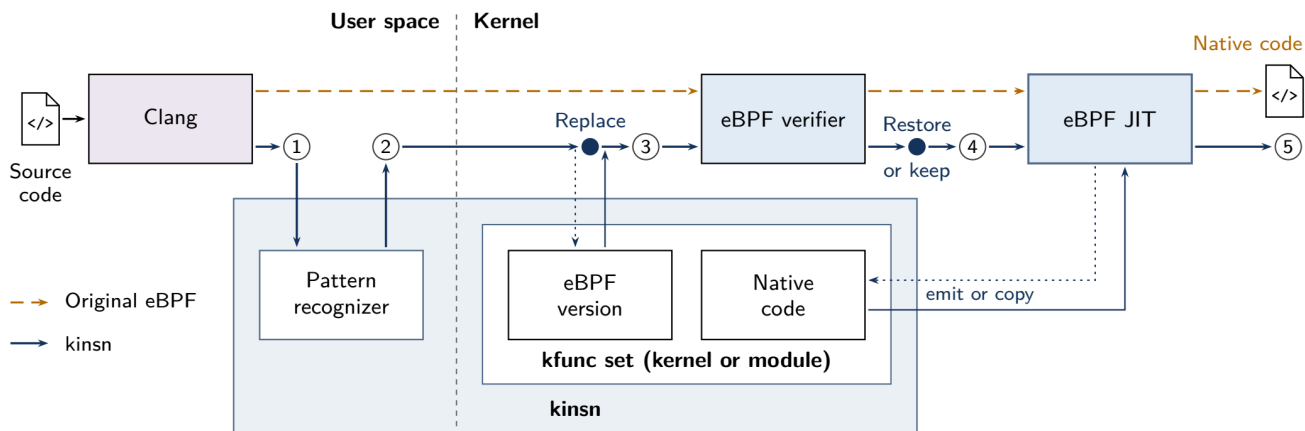
The program makes a plain kfunc call. Arguments named `__k` must be known constants.

The kfunc set registers the eBPF version next to the kfunc: ALU, loads, stores, and forward jumps that stay inside it.

The x86-64 JIT copies the compiled kfunc, or calls an emit callback for five kfuncs. Otherwise the eBPF version runs.



Check the eBPF version, run native code



Formally verify equivalence between eBPF and native

VERIFIER

The verifier checks the eBPF version at each call, like any BPF code.

JIT CHECK, X86

Copied code may not jump, use RIP-relative addressing, or divide.

REQUIRES ADDITIONAL PROOF

Formally verify they are equal: the eBPF version and the native code compute the same result.

FALLBACK

With constant blinding or verifier rewrites, the eBPF version runs.

New trusted code: a checked copier per JIT, and code for a few kfuncs.

RFC: About 690 core kernel lines: verifier and x86 JIT

```

/* include/linux/btf.h */
struct bpf_kfunc_body {
    const u32 *id;
    const struct bpf_insn *insns; /* checked */
    u32 len;
    int (*emit)(...); /* optional */
};
/* bpf_verifier.h: one per call site */
struct bpf_kfunc_inline {
    struct bpf_insn call;
    const struct bpf_kfunc_body *body;
    /* bound registers, __k constants */
    u8 reg[MAX_BPF_FUNC_REG_ARGS + 1];
    s32 imm[MAX_BPF_FUNC_REG_ARGS + 1];
    ...
};
/* filter.h: arch hook, default -EOPNOTSUPP */
int bpf_jit_inline_kfunc(
    const struct bpf_kfunc_inline *in, u8 *buf);

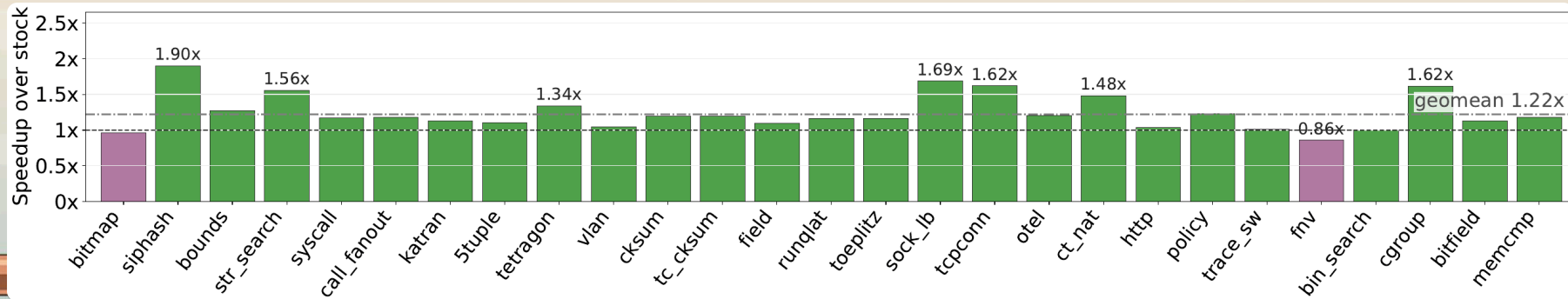
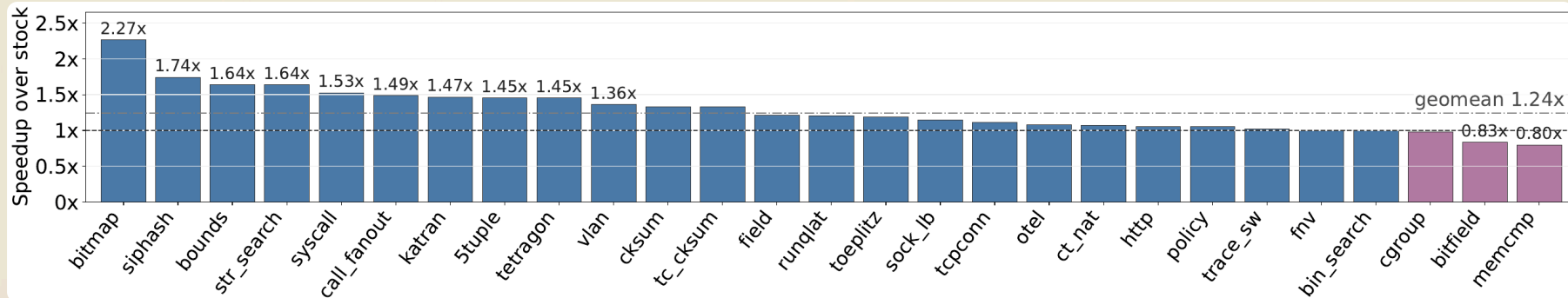
```

PLUMBERS CONFERENCE PRAGUE, CZECHIA / OCT. 3-7, 2020

FILE	+	CHANGE
kfunc_inline.c	273	new: expand, restore
verifier.c	37	calls into it
btf.c	126	register bodies
x86 JIT	181	copy kfuncs
headers, core	77	types, hooks

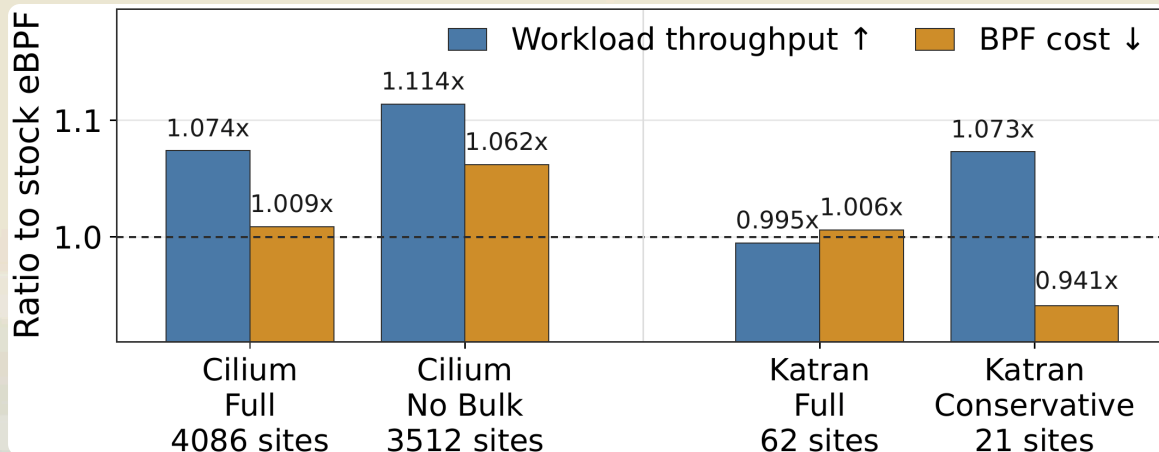
Seven kfuncs so far: rol64, select64, extract64, load_be64, lea64 (x86 emits rol, cmov, bextr, movbe, lea); prefetch, copy16 (copied).

Seven operations: 1.24× faster on x86-64, 1.22× on arm64



Top: x86-64. Bottom: arm64. Each bar is one benchmark.

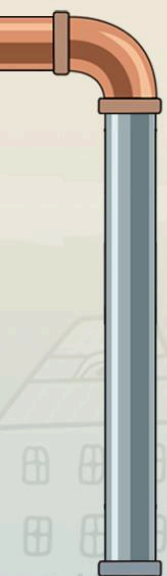
Optimization depends on the CPU and the workload



On Cilium x86-64, dropping bulk-memory operations raised throughput from 1.074x to 1.114x.

On Katran arm64, 21 chosen call sites gave 1.073x, but all 62 gave 0.995x.

More may be worse: Select instructions still require a complex cost model → in userspace LLVM



PART 2

kprog: simulate native in eBPF

Whole native programs, still judged by the verifier



LINUX
PLUMBERS
CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026

kinsn cannot fix what spans the whole program

Registers: BPF has 11; x86-64 has 16 and arm64 31. The JIT gives each BPF register one fixed native register.

Branches: code that kinsn copies cannot jump, so branches stay as the JIT emits them.

Scheduling: the JIT keeps the bytecode's instruction order.

Calls: subprograms and most helpers stay real calls.

1.074×

Cilium with kinsn, x86-64

1.38×

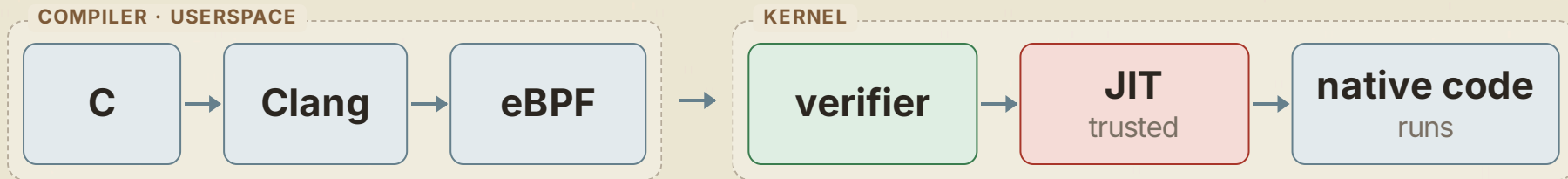
Cilium as native code, x86-64

These need the whole program compiled natively, and still verified.

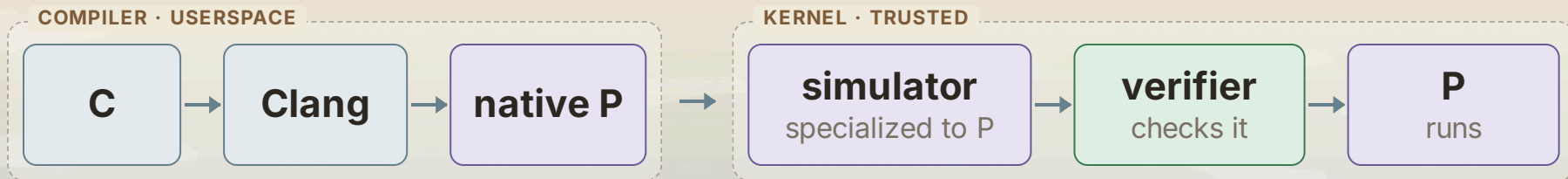


Slaying the JIT: verify a simulator, run native code

eBPF today: compile after verification



kprog: simulate before verification and run native



Trust the simulator instead of the JIT. Good news: building and formally proving the simulator is easier than proving an optimized JIT; 3000 LOC for x86 or arm.



Each native instruction becomes one simulator step

```
/* native-sim/x86/x86_sim_local_bpf.h */
struct x86_sim_state {
    X86_SIM_L_FOR_EACH_GPR(X86_SIM_L_STATE_REG_FIELD)
    __u8 cf; __u8 zf; __u8 sf; __u8 of;
    ...
};
/* one branch per native instruction */
} else if ((OP) == X86_OP_RORX) {
    __u64 __x86_l_src = X86_SIM_L_READ_REG(SRC);
    X86_SIM_L_WRITE_REG_WIDTH((DST),
        x86_ror(__x86_l_src, (IMM), __x86_l_width),
        __x86_l_width);
}
/* helper calls stay helper calls */
void *__x86_bpf_ret = bpf_map_lookup_elem(
    X86_SIM_L_READ_REG_PTR(X86_RDI),
    X86_SIM_L_HELPER_ARG_PTR(X86_RSI));
X86_SIM_L_WRITE_REG_PTR_TAG(X86_RAX, __x86_bpf_ret,
    LINUX_X86_SIM_TAG_MAP_VALUE);
```

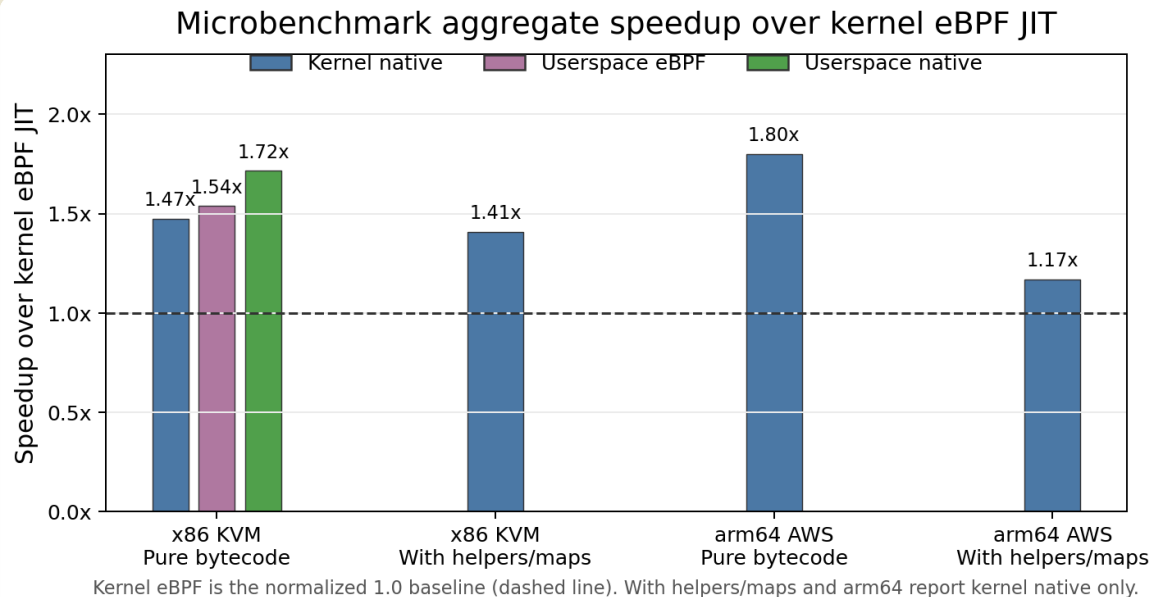
x86 registers and flags are ordinary eBPF state; pointers carry a tag.

Each native instruction is one handler in eBPF C, checked like any BPF code.

Helper calls stay real helper calls, checked as usual.

Why a JIT? Simulate it in eBPF!

Native code runs 1.47× faster on x86-64, 1.80× on arm64



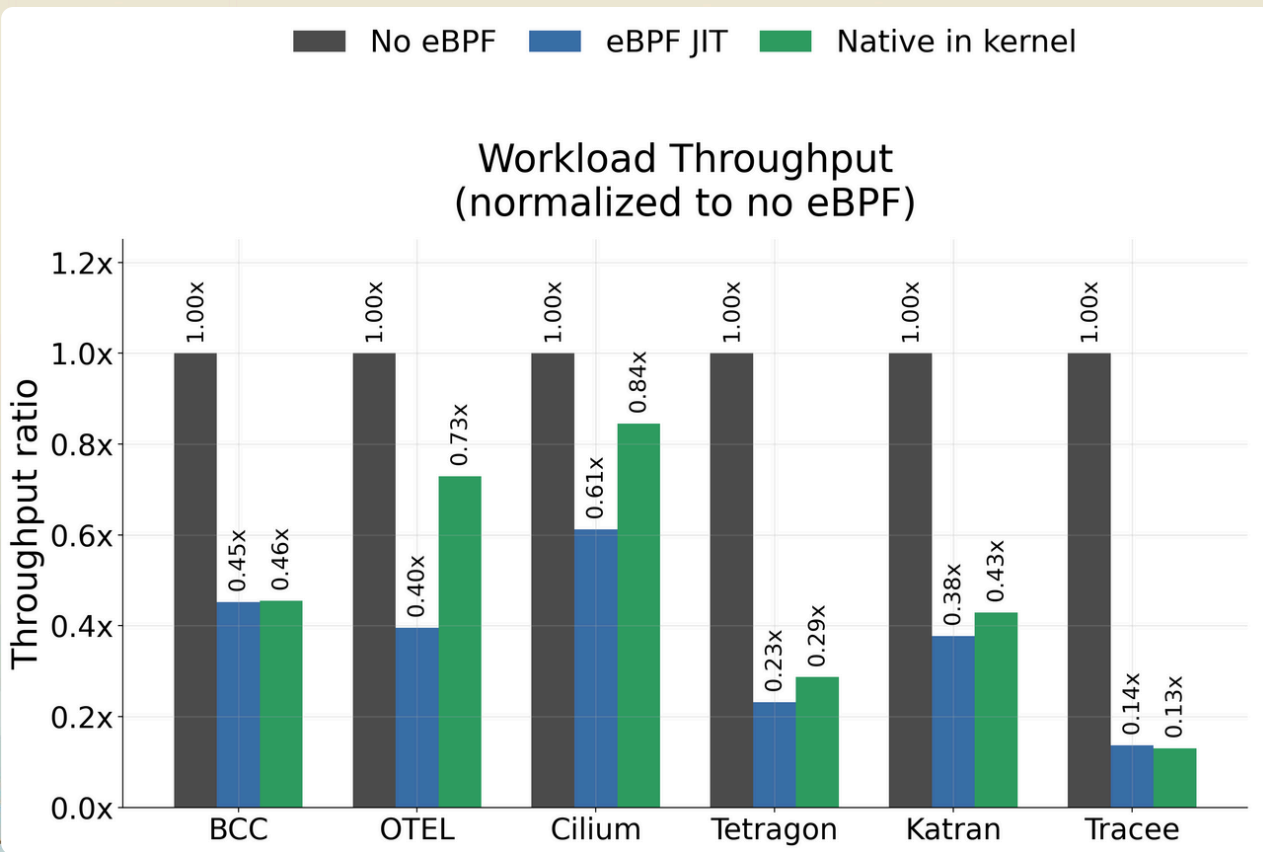
With helpers and maps, the gain is 1.41× and 1.17×.

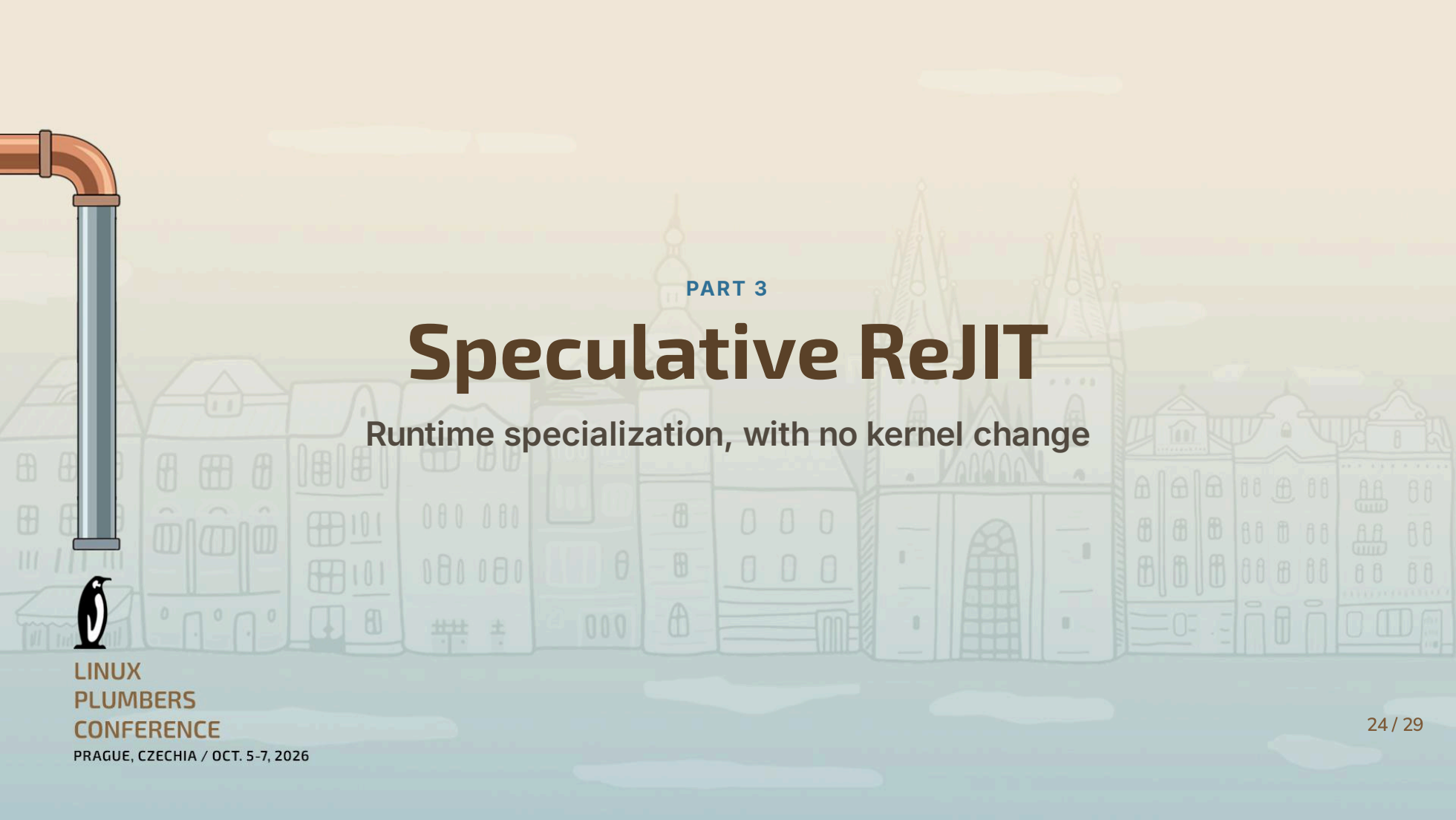
Workaround for the verifier's limit: after load, a trusted Clang daemon does the const propagation.

The cost: loading is 54× slower.

Native code helps most in compute-heavy programs.

Native code raises application throughput by up to 1.83x





PART 3

Speculative ReJIT

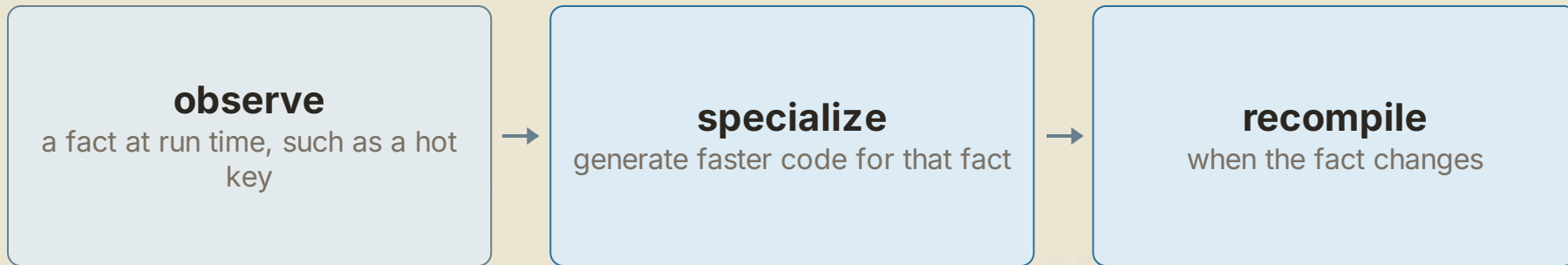
Runtime specialization, with no kernel change



LINUX
PLUMBERS
CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026

Background: speculation in JIT compilers



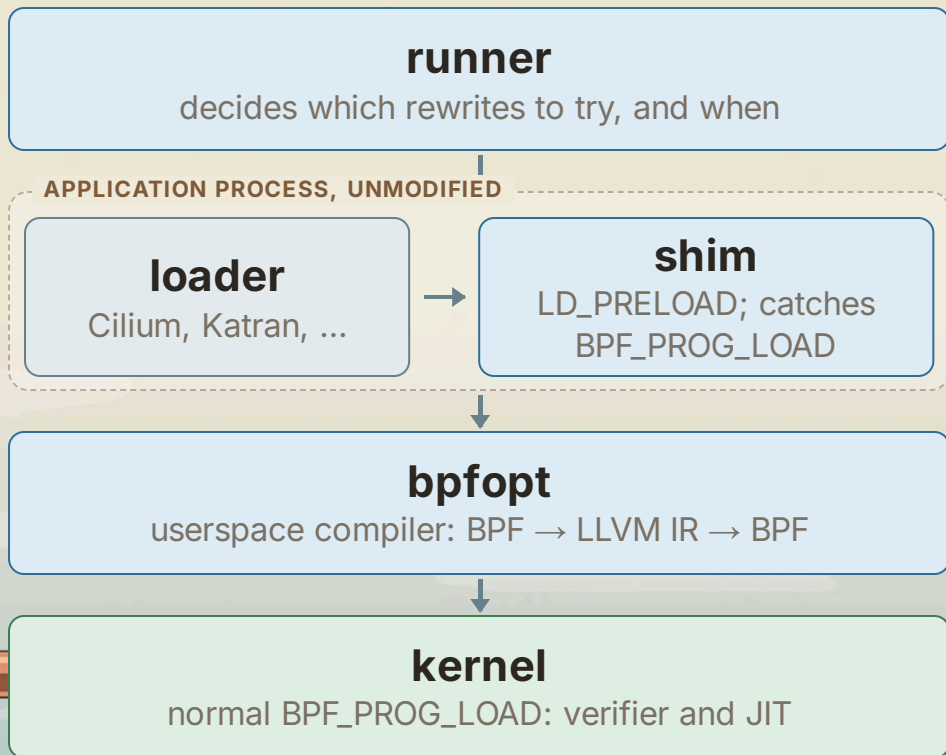
Language JITs such as V8 and HotSpot do this all the time.

The kernel's eBPF JIT does none of this: it compiles once, at load time.

Can eBPF speculate, when every program must pass the verifier?



Separate safety from correctness in userspace



Requirements: work inside unmodified loaders, pass the existing verifier, and replace programs that are already running.

Key insight: the verifier already guarantees safety, so the optimizer only has to be correct, and it can live in userspace.

The shim swaps in the rewritten program at load time, or reloads it later and updates its attachments.



Optimization passes

PASS	WHAT IT DOES
map inlining	turns stable map values into constants
bytecode rewriting	re-optimizes the bytecode with LLVM O3
hot-region versioning	clones the hot path behind one-sided branches
branch layout	puts the likely side first, from branch profiles
tail-call inline caching	adds a fast path for hot tail-call targets
context specialization	specializes for common context values

Measured: throughput 1.091× in Katran and 1.067× in Cilium; program latency -12% in Katran, -35% in BCC, -25% in Cilium, -21% in Tetragon, -26% to -54% in OTel.



kinsn and kprog trade trust for speed; ReJIT is orthogonal

	KERNEL CHANGE	NEWLY TRUSTED	SPEED (X86-64)
eBPF today	–	–	baseline
kinsn	about 1,060 lines	native code per JIT, emitted or copied	micro 1.24×, Cilium 1.074×
kprog	same as kinsn, or just use a kernel module	simulator generator and linker	micro 1.47×, apps up to 1.83×

Speculative ReJIT is orthogonal. It specializes on the workload, with or without kinsn and kprog, and in userspace.

All three keep the existing verifier as the judge of safety.



Questions for the room

kinsn design

- Is kinsn, a kfunc with an eBPF version, acceptable upstream?
- Is lowering a call before verification, and restoring it after, acceptable?
- Should the eBPF version be generated from the kfunc's C source?
- Should the JIT both hand-write and copy native code, or only copy?

Research ideas: Are kprog and speculative optimization useful?

Paper: arxiv.org/abs/2606.24213

Code:

github.com/eunomia-bpf/bpf-benchmark

