



Verifier-Free Kernel Extensions with Rust



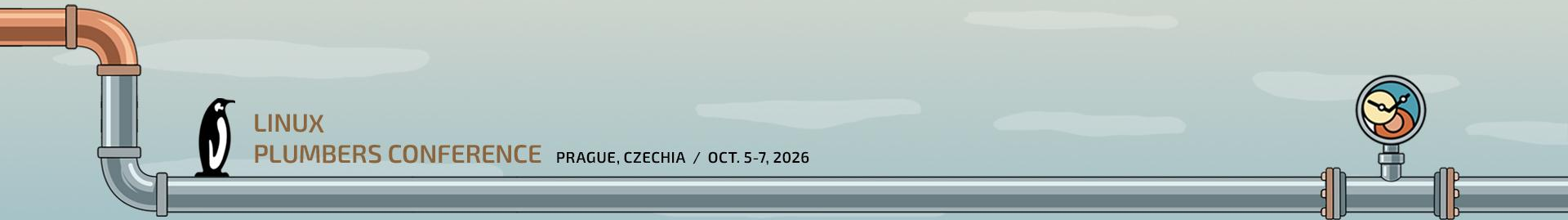
LINUX
PLUMBERS
CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026

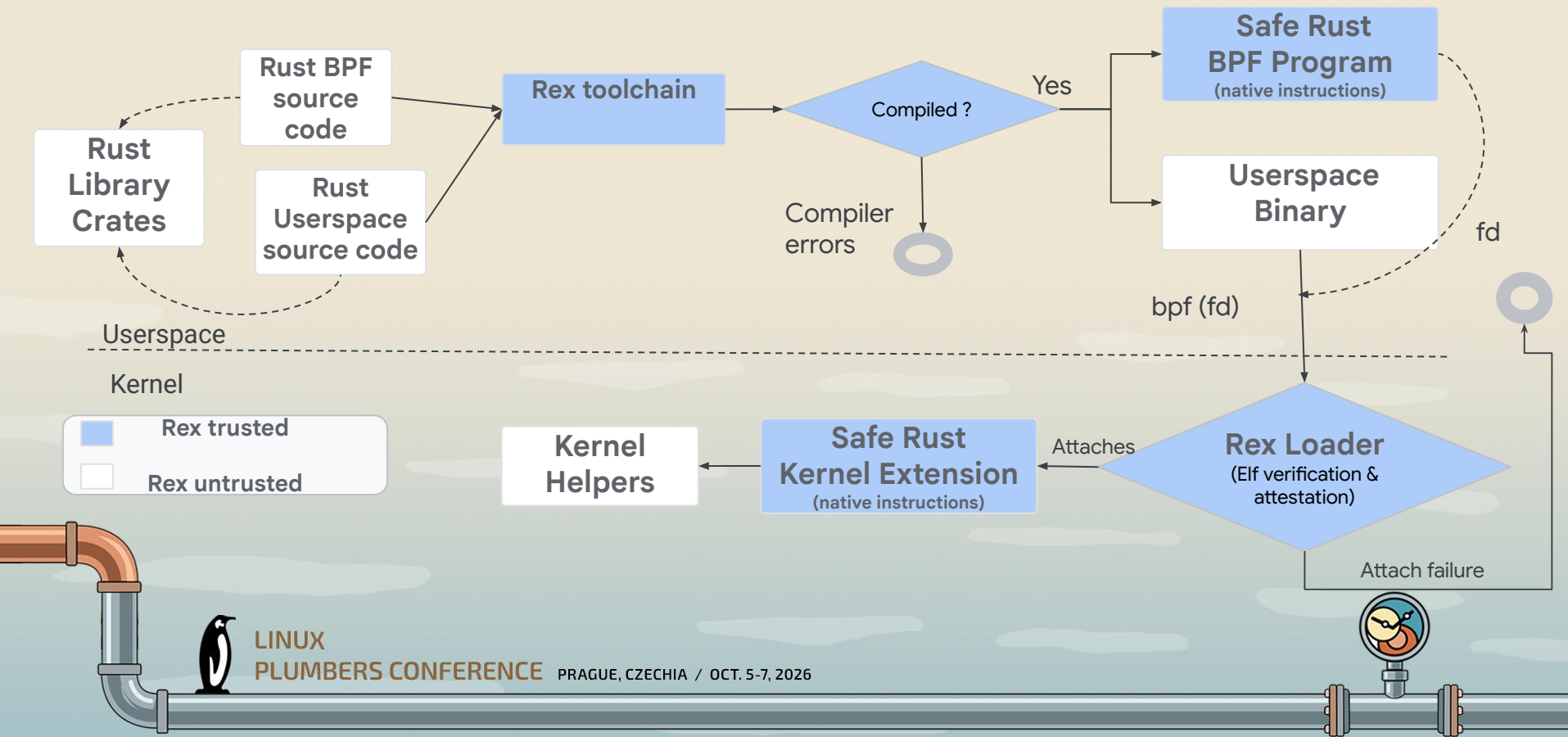
- Aniket Gattani (Google)
- Collaborators: Jinghao Jia (Google), Tianyin Xu (UIUC)

Agenda

- Vision
- Problem statement
- Design
- Native Instruction Analysis
- Performance and Statistics
- Known Gaps
- Feedback / Q&A



Vision



The background features a light-colored, stylized illustration of a city skyline. On the left, a large, vertical pipe with a blue body and orange-brown flanges extends from the top to the bottom. The city skyline consists of various buildings, including a prominent church with multiple spires and a clock tower. The sky is a pale yellow with soft, horizontal cloud-like shapes. The water in the foreground is a light blue with some darker blue patches.

Problem Statement



LINUX
PLUMBERS
CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026

Scope

- Have you spent non-trivial efforts writing verifier friendly code?
- Do you write large BPF Programs? (> 1k LoC)
- Do you care about potential speedups from native LLVM vs BPF JIT instructions? (upto 20% CPU cycles saved per extension)
- Do you care about saving some memory per-BPF program?
- Do you care about program cancellation?



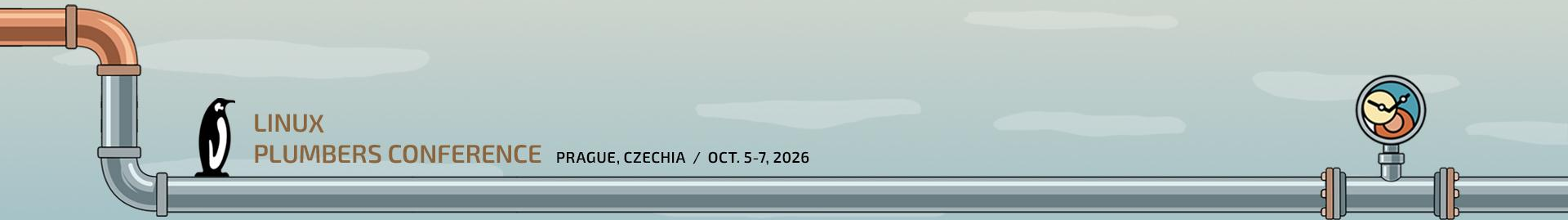
LINUX
PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



What this is not about

- Rust vs C
 - However, we do depend on Rust toolchain
- Replacing the self-sufficient BPF verifier
 - In fact we can co-exist sharing same maps
- Adversarial threat model
- Extending the compiler trust into the custom toolchain.
- Not the same as moving the verifier in user-mode Linux
 - But left shifting the safety guarantees



Design



LINUX
PLUMBERS
CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026

Rex-1.0 Pushbacks

Rex was developed as a research project by Jinghao (UIUC), Dan (Virginia Tech) & Tianyi (UIUC).

Per-cpu
stack buffer
makes
execution
non-reentra
nt

```
In-kernel dispatcher function
rex_dispatcher_func:
// save the callee-saved registers
pushq ...
// switch stack, set termination state
movq %rsp, PER_CPU_VAR(rex_old_sp)
movq PER_CPU_VAR(rex_stack), %rsp
movq ..., PER_CPU_VAR(rex_start_time)
...
// invoke the REX program
call *%rdx
rex_exit:
// reset stack and termination state
movq PER_CPU_VAR(rex_old_sp), %rsp
...
// restore the callee-saved registers
popq ...
ret
```

Rex program

```
rex_prog1:
...
ret
```

Panic

Rex panic handler

```
// cleanup resources
...
call rex_landingpad
```

In-kernel landingpad

```
rex_landingpad:
// report error
// set default return value
...
jmp rex_exit
```

Resource
tracking
penalty

Rex-2.0 Guarantees

Termination

- No recursions (single code-gen and LTO)
- Deterministic fuel checking (Wasmtime)

Custom LLVM
Pass
(427 LoC)

Rust Compiler

Rust toolchain

Memory /
Type safety

Runtime Protection

- No panics
- Stack safety

Resource safety

- RAI & Drop traits



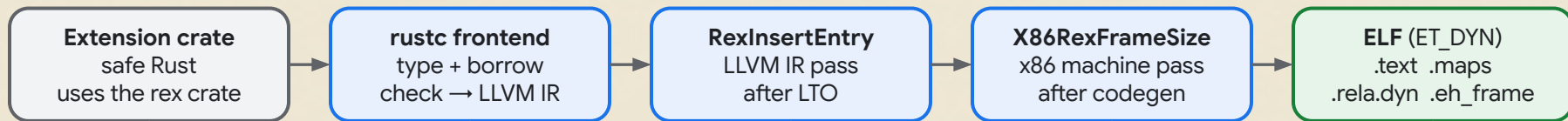
LINUX
PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026

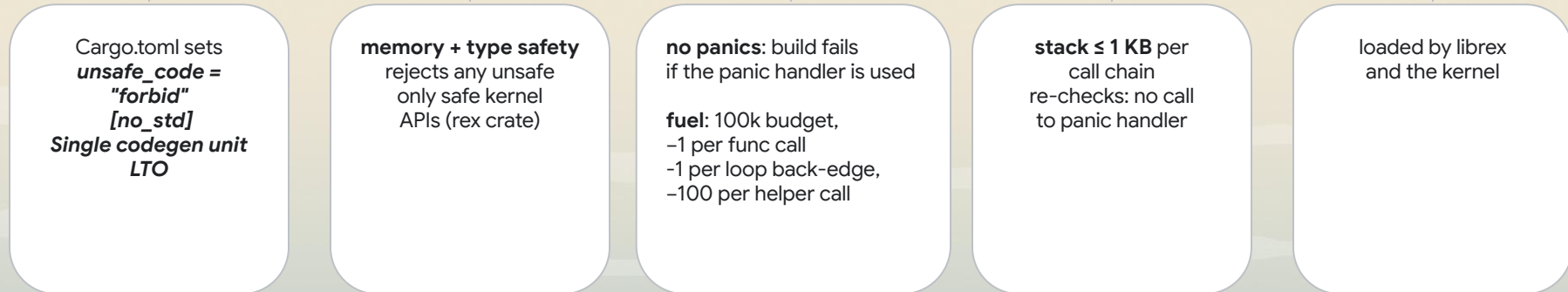


Compile Stage

Stages



Checks



LINUX

PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



Load / Attach Stage

Loader (librex + libbpf)

Kernel: bpf() syscall

parse ELF: .maps, .rela.dyn, symbols

① **BPF_MAP_CREATE** per map (librex)

map fds

ordinary BPF maps,
shareable with C BPF programs

② **BPF_PROG_LOAD_REX_BASE**

rustfd, map fds, relocs, symbols

base prog fd

kernel reads the ELF itself (from rustfd)
check header; segments aligned, no overlap, W^X
alloc exec memory, patch map pointers
apply relocations, resolve symbols via kallsyms
register function names in kallsyms
start the watchdog

loop [for each entry point]

③ **BPF_PROG_LOAD_REX**

base_prog_fd, prog_offset

prog fd

normal struct bpf_prog with no_bpf = 1
bpf_func = base image + offset
standard license / cap checks
one shared image per object (BPF: one program per entry function)

④ **BPF_LINK_CREATE** (unchanged)

existing bpf_link attach path



LINUX

PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



Termination

- Why? Because static termination != runtime termination
- Similar to bpf's `may_goto`, relaxed for loops may cause stalls holding spinlocks
- Leverage LLVM's unwind tables and LSDA
- Can achieve zero-cost unwinding
- NMI for watchdog triggers - can cancel whilst holding bpf_spinlocks.
- Not a free lunch:
 - Takes about 3kB stack using [unwinding](#) crate (can be reduced)
 - Drop traits can only be inside Rex's crate
 - Panic on unwind is not supported



LINUX
PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



Trusted Computing Base

Module	eLoC	Use case
LLVM Pass (C++)	427	Static termination / stack safety guarantees
libRex (C++)	648	libbpf equivalent
In-kernel Rex loader (C)	992	Lifecycle management and attachment of the programs
Rex Crate + Macros (Rust)	3,514 (772 unsafe lines)	FFI, Drop traits and Macros
Optional In-kernel Termination module (C)	403	Runtime Watchdog termination
Optional Unwinding Rex library (Rust)	1,605 (can be reduced since we don't need the entire crate)	Third party unwinding crate for cancellation. (2M+ downloads)



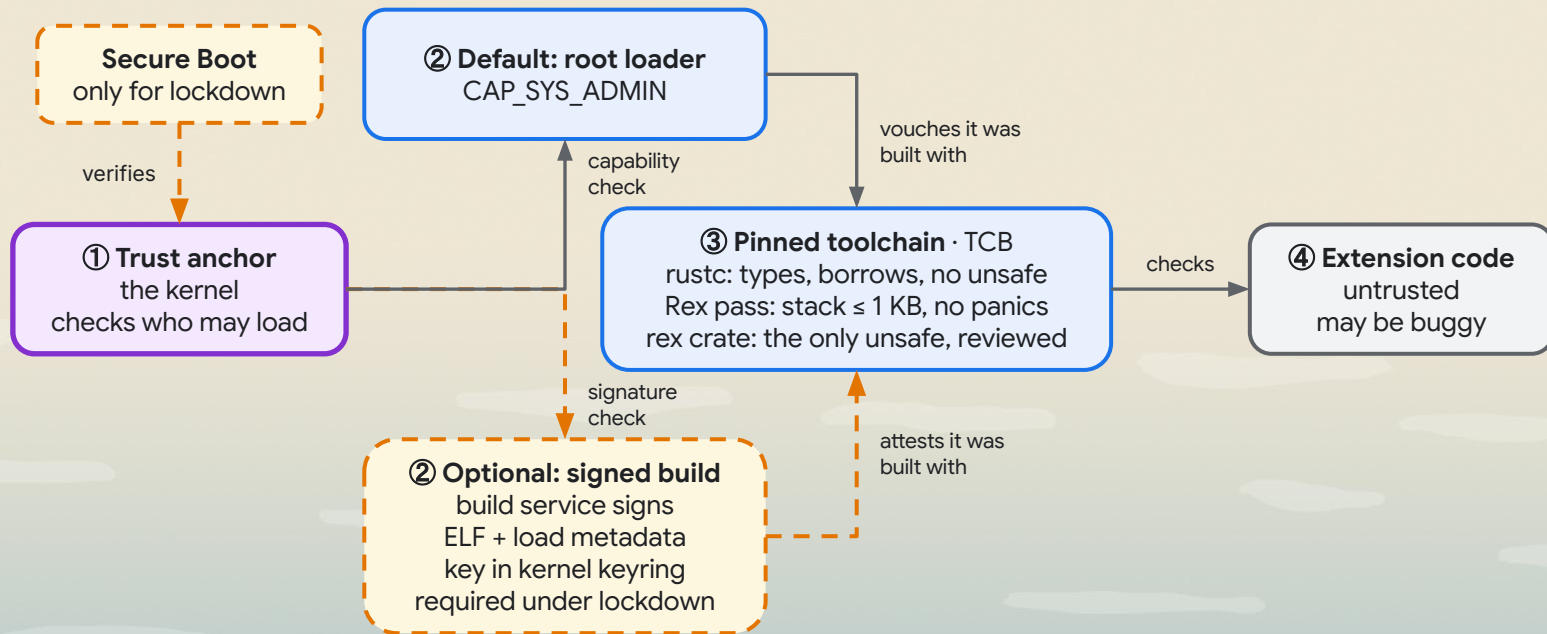
LINUX
PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



Chain of Trust

Threat Model: bugs, not malice



Kernel backstops
outside the chain

At load
ELF checks · W^X · relocations

At run time
watchdog cancels after 1 s · unwinding releases locks

Native Instructions Analysis



LINUX
PLUMBERS
CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026

Find the first set bit

BPF-C

11 to 50 cycles (De Bruijn multiplication)

```
int idle_idx = __builtin_ctzll(ctx->map);
```

```
37: movq %rdi, %rsi
3a: negq %rsi
3d: movq %rdi, %rdx
40: andq %rsi, %rdx
43: movabsq $151050438420815295, %rsi
4d: imulq %rsi, %rdx
51: shrq $58, %rdx
55: movabsq $-52576573274840960, %rsi
5f: addq %rdx, %rsi
62: movzbq (%rsi), %r14
```

Rex-2.0

1 to 3 cycles (tzcntq)

```
let idle_idx = map.trailing_zeros();
```

```
2ce1: tzcntq %r14, %r12
2ce6: orl %ebx, %r12d
```



LINUX
PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



Clear a bit of a bitmask

BPF-C

```
uint64_t i = __sync_fetch_and_and(&cpuset->idle_mask[slot],  
mask);
```

```
58d: movq %rax, %r10  
590: movq 96(%r14), %rax  
594: movq %rax, %r11  
597: andq %rdi, %r11  
59a: lock  
59b: cmpxchgq %r11, 96(%r14) # CAS loop  
5a0: jne 0xffffffffc039319c  
5a2: movq %rax, %rdi  
5a5: movq %r10, %rax
```

Rex-2.0

```
let mask = 1u64 << bit;  
let old = self.data.idle_mask[slot].fetch_and(!mask, Ordering::SeqCst);
```

```
225c: addq $0x188, %rcx ; # &cpuset.idle_mask  
2263: lock btrq $0x0, (%rcx) ; # Bit test and reset  
2269: jb 0x2179 ; # If bit was set
```



Rotate 64 bits and multiply

BPF-C

Rotate (5 cycles) + Multiplication (3 cycles)

```
_static inline __u64 rol64(__u64 x, int r) {  
    return (x << r) | (x >> (64 - r));  
}
```

```
h = rol64(h, 29) * 5 + 0x52dce729ULL;
```

```
86: movq  %r13, %rax
```

```
89: shrq  $35, %rax
```

```
8d: shlq  $29, %r13
```

```
91: orq   %rax, %r13 # 4 instructions
```

```
94: imulq $5, %r13, %r13 # integer multiplication
```

Rex-2.0

Rotate (1 cycle) + Multiplication (2 cycles)

```
h = h.rotate_left(29).wrapping_mul(5)
```

```
2cb3: rol  $0x1d, %r14 ; # Rotate in 1 cycle
```

```
2cb7: lea  (%r14,%r14,4), %rdx ; # Multiply with LEA
```



LINUX
PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



128 Bit Wide Multiplication

BPF-C

25 instrns | 4 multipliers | 15-20 cycles

```
static inline void mul128(__u64 a, __u64 b, __u64 *lo, __u64 *hi) {
    __u64 a_lo = (__u32)a, a_hi = a >> 32;
    __u64 b_lo = (__u32)b, b_hi = b >> 32;
    __u64 p0 = a_lo * b_lo, p1 = a_lo * b_hi;
    __u64 p2 = a_hi * b_lo, p3 = a_hi * b_hi;
    __u64 mid = p1 + (p0 >> 32), mid_lo = mid + p2;
    __u64 carry = (mid_lo < mid) ? 1 : 0;
    *lo = (p0 & 0xffffffffULL) | (mid_lo << 32);
    *hi = p3 + (mid_lo >> 32) + (carry << 32);
}
57: movl    %esi, %eax ; a_lo      59: movl    %edi, %r8d ; b_lo
5c: shrq    $32, %rsi ; a_hi      60: shrq    $32, %rdi ; b_hi
67: imulq   %rsi, %r14 ; p1 = b_lo * a_hi
6e: imulq   %rdi, %r13 ; p2 = a_lo * b_hi
72: imulq   %r8, %rax ; p0 = a_lo * b_lo
76: imulq   %rdi, %rsi ; p3 = a_hi * b_hi
81: addq    %r14, %r9 ; mid = p1 + (p0 >> 32)
87: addq    %r13, %r10 ; mid_lo = mid + p2
8a: cmpq    %r9, %r10 ; carry check
8d: setb    %cl ; carry flag
. . . . .
```

Rex-2.0

3 instrns | 1 hw mul | ~4 cycles

```
let prod = (a as u128) * (b as u128);
let k3 = (prod as u64) ^ ((prod >> 64) as u64);

57: movq %rsi, %rax ; %rax = b
5a: mulq %rdi ; [%rdx:%rax] = a * b
5d: xorq %rdx, %rax ; lo ^ hi (1 cycle)
```

Performance and Statistics



LINUX
PLUMBERS
CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026

Testing Methodology

- We battle-tested with a production-grade VM scheduler (sched_ext)
 - Highly optimized CCX-aware policy ([LPC 2025](#)).
 - A scheduler is a real test of large kernel extension since it is sensitive to throughput, cache, latency and wait-times.
 - Ported ~8k LoC and successfully ran on multiple benchmarks with no regressions
 - We can co-exist with BPF. (some parts in Rex, some in BPF)
- We also tested Memcached across several core counts
- Microbenchmarks hammering certain instructions



LINUX
PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



Scheduler Footprint

Memory Footprint	BPF-C	Rex-2.0	Difference	Comment
Executable (execmem)	168 KiB (42 pages)	104 KiB (26 pages)	-38%	One binary across all entry funcs
Program Bytecode / Metadata	280 KiB (70 pages)	16 KiB (4 pages)	-94%	No un-JITed code
Total	448 KiB (112 pages)	120 KiB (30 pages)	-73%	

Code Footprints	BPF-C	Rex-2.0	Difference
sLoC	7,710	7,983	+273 (+3.5%)
Lexical Tokens	53,043	50,847	-2,196 (-4.1%)
Logical Statements (Semicolons)	3,674	3,135	-539 (-14.7%)

Noise
Gains
Regression



LINUX
PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



Subprog Stats

Load time reduced from **495 ms** to **14ms**

Attach Points	Stack Usage (Bytes)		Budget		Runtimes (ns)	
	BPF-C	Rex-2.0	BPF-C (instrns)	Rex-2.0 (Fuel)	BPF-C	Rex-2.0
select_cpu (select cpu on wakeup edge)	176	272	70,876 (~7%)	1,560 (~1.6%)	609.1	425.21 (-30.2%)
msg_send (task lifecycle events & userspace msgs)	216	464	216,501 (~22%)	3,805 (~4%)	259.1	263.8 (+1.8%)
dispatch (pick next task on a cpu)	368	672	374,606 (~37%)	18,123 (~18%)	597.2	472.8 (-20.8%)

Noise
Gains
Regression



LINUX
PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



5% Overcommit (Antagonists)

Benchmark (ns)	Metric	BPF	Rex-2.0	Difference	Sensitive To:
RT Bench	p50 (μs)	41.8	41.5	-0.7%	Wakeup Latency and wait time
	p99 (μs)	80.4	79.5	-1.1%	
	p99.9 (μs)	126.2	126.25	0.0%	
	p99.99 (μs)	971.6	872.25	-11.4%	
SpecjBB	critical_jOPS	21,514.75	21,947.33	+1.97%	Cache locality and wait times
	max_jOPS	43,033.25	43,732.33	+1.6%	
Multichase	LdAvg (Mib/s)	46,348.18	46,209.64	-0.30%	Cache locality and memory bandwidth
	LdMax (Mib/s)	63,294.13	62,994.32	-0.48%	
	Latency (ns)	9.154	9.194	-0.43%	
SIR-17	score	68.9	68.85	-0.07%	Migrations and lost cpu cycles

Noise
Gains
Regression

LINUX
PLUMBERS CONFERENCE PRAGUE, CZECHIA / OCT. 5-7, 2026



Memcached (ops/s)

# of Cores	Vanilla Memcached	BPF-C	Rex-2.0	Rex-2.0 vs. Vanilla	Rex-2.0 vs. BPF
1	326,714	558,070	588,251	+80.1%	+5.4%
2	534,601	869,486	930,002	+74.0%	+7.0%
4	879,914	1,558,797	1,650,931	+87.6%	+5.9%
8	1,055,945	2,737,004	2,888,145	+173.5%	+5.5%
16	1,763,048	4,251,376	4,311,292	+144.5%	+1.4%
32	1,613,579	6,841,937	6,916,079	+328.6%	+1.1%
64	6,315,305	9,579,637	9,685,368	+53.4%	+1.1%
128	10,122,028	12,109,385	13,170,100	+30.1%	+8.8%



LINUX
PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



Microbenchmarks

Benchmark (ns)	BPF-C	Rex-2.0	Difference	Key Microarchitectural Driver
Packet Encapsulation (lower)	1.70	0.91	-46.5%	Store coalescing (movl/movq) vs byte-by-byte BPF stores
RFC 1071 Checksum (lower)	6.79	3.95	-41.8%	adcq carry chains vs branch-heavy BPF emulation
Flow Hash (lower)	6.54	5.24	-19.9%	(rolq / mulq) vs multi-instruction BPF synthesis



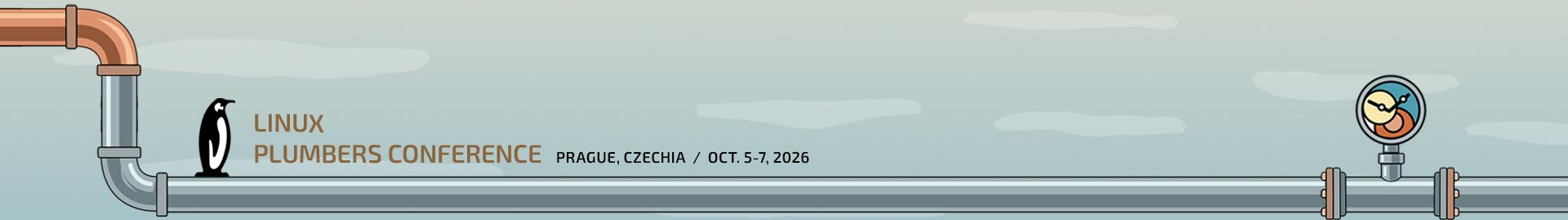
LINUX
PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



Known gaps

- No co-re / arena support / Tail calls
- BPF & Rex-2.0 can co-exist but need a stronger API for map layout
- Spectre V1 mitigation
- Kernel-pointer leaks to userspace
- New features since 6.18 kernel
- Kernel - Compiler version pinning need to be fixed
- A bug in compiler pass implies recompiling and resigning the binaries alongwith kernel patch
- Compiler versioning x Kernel versioning can become nightmare to maintain
- Evaluation on attack vector from third-party crates



Feedback / Q&A



LINUX
PLUMBERS
CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026

Programmability Examples



LINUX
PLUMBERS
CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026

Needless compiler Barriers

/ 1. A linked list over an array map: indices, not pointers */*

```
struct arr_list { u32 first, last; };  
struct arr_list_entry { u32 next, prev; };
```

/ 2. Keep clang from optimising the bounds check away: the*

** verifier only trusts the register it saw being compared */*

```
#define MUST_CHECK(x) ({ asm volatile("" : "+r"(x)); x; })
```

```
#define id_to_elem(arr, n, id) ({  
    typeof(&arr[0]) ret = NULL;  
    if (id) {  
        u32 idx = id - 1;  
        if (MUST_CHECK(idx) < (n))  
            ret = &(arr)[idx];  
    }  
    ret; })
```



LINUX

PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



for-loop Boilerplate

finding idle cpu within a list of 64 bit bitmasks

BPF-C

```
static int noline find_idle_cpu_cb(u32 i, void *data) {
    struct idle_ctx *ctx = data;
    if (!ctx->map) return 1;    /* re-check, for the verifier */
    int bit = __builtin_ctzll(ctx->map), cpu = ctx->slot * 64 + bit;
    ctx->map &= ~(1ULL << bit);

    ...
    pcpu->scratch.idle_candidate = cpu;    /* scratch space for return value
to prevent extra stack usage */
    return 1;
}

/* caller - itself the callback of an outer bpf_loop() */
ctx->map = mask[i]; ctx->slot = i;
bpf_loop(64, find_idle_cpu_cb, ctx, 0);
```

Rex-2.0

```
for (i, slot) in pool.idle_mask.iter().enumerate() {
    let mut map = slot.load(Ordering::Relaxed);
    while map != 0 {
        let bit = map.trailing_zeros();
        map &= map - 1;
        let Some(cpu) = Cpu::new(i * 64 + bit) {
            return Some(cpu);
        }
    }
}
None
```



LINUX

PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



Register Clobbering

BPF-C

```
int wakeup(struct task *t, u64 now, int waker_cpu)
{ ... } /* global: verified alone, never reads waker_cpu */

int wakeup_wrapper (struct task *t, u64 now, int waker_cpu) {
    /* waker_cpu arrived in r3 */
    ...

    s = bpf_map_lookup_elem(&scheds, &k); /* helper: clobbers r1-r5 */
    // asm volatile("" : "+r"(waker_cpu)); /* force clang to keep the register alive */
    wakeup(t, now, waker_cpu); /* <- R3 is not a scalar */
}
```

Verifier Error log

```
call bpf_map_lookup_elem ; R1-R5 are caller-saved: R3 = <not init>
                                ; arg 3 is dead in wakeup, so clang passes
                                ; poison for it: r3 is never written again
call wakeup#1203 ; global fn: every argument register checked
```

R3 is not a scalar



LINUX
PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



Resources

- <https://lwn.net/Articles/964381/>
- <https://lore.kernel.org/all/20250304003239.2390751-1-memxor@gmail.com/>
- <https://lwn.net/Articles/969185/>
- <https://lwn.net/Articles/938435/>
- <https://lwn.net/Articles/1075067/>

