

rust-bpf updates

Alexei Starovoitov (Linux Plumbers, Oct 2026)

THE GOAL

If rustc accepts it, the verifier accepts it

Upstream rustc. core and alloc as they are.

BTreeMap, Box, dyn Trait, Drop, panic!()
have to work.

rustc finds programmer's mistakes in user
space. The verifier stays in the kernel as
the security boundary.

bpf list, Sep 2026

Accept-all-rust doesn't mean accept
rust that can leak resources.

Locks, references, RCU, types of kfunc arguments are
still checked. Obnoxious unsafe and inline asm will
be rejected.

WHAT IT LOOKS LIKE

scx_simple in Rust

```
static STATS: BpfCell<Option<
    BTreeMap<&str, AtomicU64>>> = ...;

trait SchedPolicy {
    fn enqueue(&self, p: &TaskRef,
               vtime: u64, flags: u64);
}

impl SchedPolicy for FifoPolicy { ... }
impl SchedPolicy for VtimePolicy { ... }

fn stat_inc(key: &'static str) {
    let stats = unsafe { &*STATS.0.get() };
    stats.as_ref().unwrap()
        .get(key).unwrap()
        .fetch_add(1, Relaxed);
}
```

```
bpf_prog!("struct_ops/simple_enqueue",
fn simple_enqueue(p: *mut task_struct,
                  enq_flags: u64) {
    let p = TaskRef(p);
    stat_inc("global");
    let vtime = p.scx_dsq_vtime();

    let policy: &dyn SchedPolicy =
        if FIFO_SCHED.load(Relaxed) {
            &FifoPolicy
        } else {
            &VtimePolicy
        };
    policy.enqueue(&p, vtime, enq_flags);
});
```

2026 SO FAR

What had to change in the kernel

Static stack liveness	Eduard Zingerman, Alexei Starovoitov	Apr	merged
More than 5 arguments	Yonghong Song, Puranjay Mohan	May	merged
Call depth 8 to 16 frames	Alexei Starovoitov	Jun	merged
Verifier errors with source, cause and suggestion	Kumar Kartikeya Dwivedi	Aug	merged
Return and pass struct and i128 by value	Yonghong Song	Sep	merged
Stack 512 bytes to 2 KiB	Kumar Kartikeya Dwivedi	Sep	merged
Indirect calls: callx	Alexei Starovoitov	Sep	merged
SCEV: widen bounded loops	Eduard Zingerman	v1	on the list
All memory in arena, BTF of Rust programs	Alexei Starovoitov	v1	on the list
panic=unwind: bpf_unwind() and landing pads	Yonghong Song	v8	on the list
New main pass. No 1M insn limit	Alexei Starovoitov	RFC	not posted

Static passes before the main pass

CFG, postorder, SCC	merged
Callees of callx	merged
Argument tracking, stack liveness	merged
Constant folding, dead branches	merged
Topological order of subprogs	merged
Live registers, 32-bit halves	merged
Dominators, loop hierarchy, SCEV	on the list
Weak topological order, relevance	RFC
Main pass: do_check()	

Stack liveness · Eduard Zingerman, Alexei Starovoitov

Was: read and write marks on verifier states, collected path by path.

Now: a forward pass finds which frame and offset a register points to. Then backward data flow per subprog and call site. 4 byte slots.

5 more passes. Loads got faster and use less memory.

Load time is the main pass. Whatever makes it do less is a win.

Calls the way rustc emits them

5+

arguments

The rest go on the stack through r11. x86, arm64, riscv64.

R0:R2

16 bytes by value

struct, union, i128 are returned in two registers and passed in two. LLVM 23.

16

call frames

Was 8. The panic path of scx_cosmos is 12 deep.

2 KiB

of stack

Was 512 bytes. x86 and arm64. cosmos_exit_task needs 544.

rust-bpf is using i128 here and there and we cannot change standard rust crates to shift arguments to satisfy JITs.

callx: dyn Trait works

```
policy.enqueue(&p, vtime, enq_flags);
```

```
// vtable in .data.rel.ro
```

```
+0    drop_in_place
```

```
+8    size
```

```
+16   align
```

```
+24   enqueue
```

```
r6 = vtable ll
```

```
r5 = *(u64 *) (r6 + 24)
```

```
callx r5
```

Static subprogs only. The callee is verified in the context of the caller, like a direct call.

All callees are known before the main pass.

Pointers are mixed with data. libbpf stores the offset of the function, the kernel guesses pointers by that value. Wrong guess is safe.

After JIT the offsets become addresses.

No change in rustc or LLVM.

SCEV: widen bounded loops

```
for (r0 = 0; r0 < 10; r0++) {  
    ...           // once, r0 in [0, 9]  
}  
  
do {  
    bytes[off] = 1;  
    i += 1;  
    off += 2;    // off in [0, 14], step 2  
} while (i < 8);
```

Dominators and loop hierarchy, then an expression for every variable that changes in the loop. The trip count gives the range.

243

of 7152 programs take fewer
insns. 11 take over 5% more

460K → 1.6K

pyperf600_nounroll, with the
follow-up

Not widened. Every iteration is walked as today

Counter or bound that is not a constant at loop entry

32-bit counters and compares

Several back edges, irreducible loops

`i *= 2`

VERIFIER

Rust programs in the main pass today

0 of 13

non-trivial programs of scx_simple and scx_cosmos load.

All fail the same way

The sequence of 8193 jumps is too complex

BTreeMap key search		7
BTreeMap descent to a leaf		2
Compare of &str keys		2
core::fmt on the panic path		2

The bound is node.len or node.height, loaded from arena. SCEV has no constant to start from. scx_simple.o is 2052 insns, the verifier gives up after 59K to 221K.

Verify as a fixpoint over the CFG

bpf-next

Walk every path of the program.

Prune a path when its state is within a state seen at that insn.

Backtrack to find which numbers have to be precise.

The work is not bounded by the size of the program. Hence 1M.

RFC

Visit insns in weak topological order. Join the states that reach an insn over different paths.

Two states are joined as a whole, and only when no later check can tell the join from the two. Otherwise both are kept.

Widen at loop heads. Values that differ by a constant from one iteration to the next keep a step, so a compare on the counter bounds the pointer that advances with it.

SCEV runs first and may replace steps in a few cases.

State pruning, precision tracking and BPF_COMPLEXITY_LIMIT_INSNS are removed.

The verifier doesn't prove that loops end

Never exits

Rejected

Walked to its end

Nothing to add. The walk is the proof

`may_goto`, `iterator`, `bpf_loop()` inside

Nothing to add

Accepted because the state converged

The verifier adds `may_goto` at the loop head

250 ms

`may_goto` takes an exit that the loop already has and the program goes on. No `bpf_throw()`, so locks and references held in the loop are fine. The timeout is counted in `bpf_prog_info` and reported in `bpf_stream`. What to do next is up to user space.

veristat on selftests and Rust

Set	Progs	Insns, bpf-next	RFC	Time, bpf-next	RFC
top	3200	5,570,496	786,996	3.58 s	2.61 s
no_alu32	3187	6,482,440	867,351	3.65 s	2.75 s
cpuv4	3201	5,552,255	779,012	3.87 s	3.05 s
libarena	136	2,412,343	1,898,448	2.58 s	2.77 s

0 → 6 of 13

Rust programs load. The other 7 fail in
core::fmt: recursion 5, buffer on the stack 2

+11213 −3122

lines in kernel/ and include/. The verifier
doesn't get smaller

1011 → 0

backtrack.c. states.c 1624 → 808. verifier.c
22755 → 27295

SIDE EFFECT

Bugs in bpf-next found on the way

Timed may_goto with private stack	Count and timestamp land on the kernel stack	fixed
Insn moved down inside its own patch	Backward call lands short of its target	fixed
may_goto +32763	Jumps backward when it expires	fixed
Iterator destroyed and created again in its loop	A program that never ends is accepted	fixed
Constant blinding of a far jump	16-bit offset overflows	fixed
may_goto budget is per function invocation	N calls run for N x 250 ms	open
may_goto looks at the clock every 65535 trips	A loop with a very long body is not bounded	open

All memory is arena

Global data in array maps

```
r1 = *(u64 *) (r1 + 0) ; scalar  
r0 = *(u32 *) (r1 + 8)  
R1 invalid mem access 'scalar'
```

A reference is an address. It can be stored in data.

No address spaces in Rust. No `__arena`, no `cast_kern`.

LLVM drops bounds checks that it proved. The verifier can't check such access to a map value.

Verifier · BPF_F_ARENA_SCALAR

```
w12 = w3  
r2 = *(u64 *) (r12 + 0)  
// arena base + low 32 bits
```

Load, store, atomic through a number is an arena access. As safe as `PTR_TO_ARENA`.

It's a flag. C keeps the error.

libbpf · .arena.data

```
#[used]  
#[link_section = ".arena.data"]  
static DATA_IN_ARENA: u8 = 0;
```

`.data` `.bss` `.rodata` go into arena. Pointers stored in data become arena addresses.

`.data.rel.ro` and `.rodata.str*` stay frozen maps: vtables for `callx`, strings for `kfuncs`.

A number is an address in arena. The JIT bounds the access at run time.

Small things that reject every Rust object

In the object	The kernel said	With the series
<pre>r3 &= 1 // core::fmt tags pointers r2 >>= 1</pre>	bitwise operator &= on pointer prohibited	The result is a number.
<pre>STRUCT 'NonNull<str>' FUNC 'write_fmt<scx_cosmos::BpfStream>'</pre>	Invalid name	Any printable character. '_' instead of it in kallsyms.
<pre>FUNC 'unwrap_failed' linkage=static '(anon)' type_id=72</pre>	Invalid arg#1	Arguments of static functions need no names.
<pre>STRUCT 'BpfCell<...>' size=32 VAR '..TASK_CTXS.0' in .bss size=1</pre>	Invalid size	LLVM splits a static into pieces. A piece is smaller than its type.

PANICS

panic=abort is not an option

unwrap(), a[i], x / y, Box::new(). Panic paths are everywhere.

bpf_throw() discards the frames. The verifier rejects it when a frame holds a lock, RCU or a reference.

Drop is the cleanup. It needs a place to run.

bpf_unwind() runs landing pads

```
fn foo() {  
    let _g = RcuReadGuard::new();  
    may_throw();  
} // Drop: bpf_rcu_read_unlock()  
  
    call bpf_rcu_read_lock  
1: call may_throw  
2: ...  
6: call bpf_rcu_read_unlock // landing pad  
    call bpf_unwind_resume  
  
.bpf_cleanup: (1, 2, 6)
```

LLVM 23 emits the table. 12 bytes per invoke region: begin, end, landing pad.

main → A → B → C. C unwinds, A has a pad

C	r0 = 0; exit after its bpf_unwind()
B	its epilogue. Puts back r6-r9 of A
A	its pad. Drops what A holds. Resume is an exit
main	its epilogue. Returns 0 to the kernel

bpf_unwind() rewrites return addresses on the stack and returns. Every frame runs its own pad on its own stack. The verifier follows the same path.

Cleanup pads only, no catch_unwind. x86 with ORC and arm64. No private stack. No second unwind in a pad.

RUSTC

From .rs to .o. No forked compiler

<code>rustc</code>	Unmodified. Stops at LLVM IR. core and alloc are built from rust-src	upstream
<code>target .json</code>	bpfel-unknown-none has no atomics, no cpu v4, no panic=unwind	custom
<code>llvm-link</code>	One module. scx_simple pulls in 28 of 536 functions of core	upstream
<code>bpf-postproc</code>	#[btf] field accessors become the IR that clang emits for CO-RE	custom
<code>opt -O2</code>	Internalize all but the entry points. 47 functions become 32	upstream
<code>add_ksyms.py</code>	Edits IR as text. Every Rust-ism that BPF can't represent is fixed here	custom
<code>llc -mcpu=v4</code>	Stock BPF backend. The object is loaded by libbpf, bpftool, veristat	upstream

Step by step, by Mykola Lysenko: mk21.substack.com/p/rust-bpf-baby-steps

What the python script papers over

What breaks

Workaround

kfuncs	rustc emits no type info for extern declarations. libbpf checks prototypes against kernel BTF.	Copy prototypes from vmlinux BTF into debug info
memcpy	A slice carries its length at run time. BTreeMap insert, copy_from_slice, == on slices can't be unrolled.	Call bpf_arena_memcpy(), bpf_arena_memcmp(). Not in bpf-next
BTF linkage	Every function is global in BTF. The verifier checks it alone, against a signature that opt has changed.	Mark internal functions as local
unreachable	A panic path ends in a call that never returns. The verifier wants exit or jump as the last insn of a subprog.	unreachable becomes ret
cold	core marks panic paths cold. llc puts them in .text.unlikely. and libbpf refuses to open the object.	opt --force-remove-attribute=cold
Landing pads	LLVM older than 23 can't emit them for BPF. The kernel side is not merged.	invoke becomes call. Both schedulers are built this way

RUSTC

Rust on BPF has two homes

Aya

Loader in Rust, aya-ebpf crates, bpf-linker. If you write BPF in Rust today, you are on it.

rust-bpf

libbpf is the loader, kernel BTF is the source of truth, same LLVM backend as C. Parity with C is the goal.

CO-RE field queries in rustc: RFC 3966, rust#161107	Michal Rostecki	open
#[btf_tag] for BTF type and decl tags	Yonghong Song	open
CO-RE type and enum relocations		missing
Atomics, cpu v4, panic=unwind in the upstream target		missing
Type info for extern functions		missing
vmlinux.rs: kernel types and kfuncs generated from BTF		missing

RUSTC

Rust in BPF selftests

On the list: first .rs file in selftests/bpf

```
$ rustc --target bpfel-unknown-none \
  -O -C panic=abort -C debuginfo=2 \
  --emit=llvm-bc data_in_arena_rust.rs
$ clang --target=bpf -mcpu=v4 \
  -c data_in_arena_rust.bc
```

Upstream rustc 1.87+ and its own target. clang 23+. Nothing is done to the output.

core is built from source in 5 seconds. No rustc on the machine: the test is skipped.

No kfuncs, no atomics, no panics in it.

Mykola Lysenko: selftests translated to Rust

577

objects build with rust-bpf

550

pass the kernel's own
prog_tests in QEMU

Prototypes of kfuncs copied from vmlinux and module BTF
unlocked iterators, module kfuncs, xdp kfuncs.

NOT DONE

Open questions in the kernel

Allocator

bpf_alloc() and bpf_free() are placeholders. kfuncs or libarena in Rust?

Stack

Arrays on the stack indexed without a check. Move address-taken locals to arena?

Recursion

core::fmt is recursive in the static call graph. Allow it when the main pass bounds it?

Summary

	Done	Upcoming
Verifier	Static passes, stack liveness, callx, 16 frames, 2 KiB stack, 16 byte values, readable errors	SCEV, arena through numbers, bpf_unwind(). New main pass without the 1M limit
rustc	Upstream rustc, core and alloc work as they are	CO-RE, atomics in the target, type info for externs, vmlinux.rs

github.com/4ast/rust-bpf