BPF in the Agentic Era

Evolving the BPF verifier and runtime to support Rust programs natively
— unlocking agent-driven development at kernel speed.

Agent-Driven Coding Loops

Agents excel when given tight feedback loops

```
write code  →  interpret compiler errors  →  fix the code

run code  →  interpret runtime results  →  write code
```

Compiler diagnostics, on-the-spot execution create tight signal loops that make agents effective.

Agents are really good at tedious things — setting up tests, reading complex error messages, booting VMs, running verifier iterations

BPF Ecosystem is Not Ready

Slow iteration

Need to boot VM, pass compiled BPF prog into VM, run veristat to get verifier log. Agents can't iterate fast.

Unreadable errors

Multi-MB verifier logs that dump everything examined. Hard for humans, harder for agents.

"Fundamental limits"

Agents hit verifier rejections and treat them as platform limits. They give up instead of working around.

- Team A found Claude "not good" at BPF C
- Team B rejected BPF as "not in-distribution."
- Company C prefers "rex" over BPF due to verifier struggles

BPF Hard Line

Crashing the kernel is never an option

- BPF doesn't trust userspace, doesn't trust language, doesn't trust compiler
- BPF analyzes assembly in the kernel

- Rust is a safe language that relies on run-time panic
  - User space -> panic=unwind
    - print reason, call drop()s, exit
  - Kernel (rust-for-linux) -> panic=abort
    - print reason, BUG()

- Rust->BPF with panic=unwind
    - print reason, call drop()s, exit from program via bpf_throw()

Isn't Plain Rust Safe for Kernel?

Rust's safety guarantees don't cover everything in kernel context

Integer overflow

```
let x: u8 =
 core::hint::black_box(255);
let y = x + 1;
```

Bounds check

```
let v = [1, 2, 3];
v[val as usize]
```

String slicing

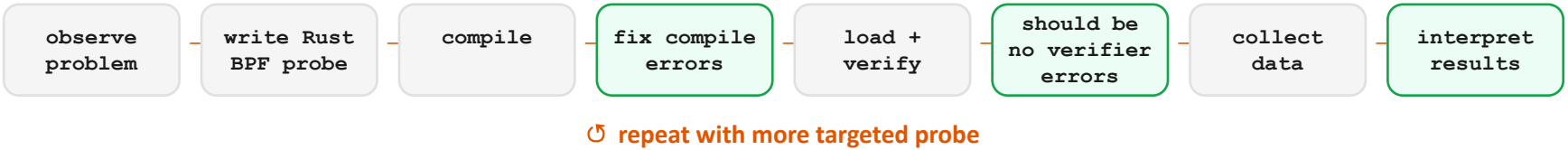
```
let s = "héllo";
&s[2..]
```

With Rust-BPF, dynamic checks don't panic the kernel. panic=unwind mode releases resources and program terminates with bpf_throw().

The verifier catches errors statically where possible. When it can't prove safety, it falls back to runtime checks instead of rejecting. Agent-written code is often correct out of the box — verifier rejection is a verifier "bug".

Agent loop with Rust BPF

Every step gives the agent structured, actionable feedback



Addressing the Gaps

Fix the toolchain

- Enable BPF in User Mode Linux, link with veristat
- "Unprivileged veristat" integrated into the compilation process
- No more booting VMs to check verifier output
- No need to say 'yes' to agent to run 'sudo'

Redesign the verifier errors

Replace full log dump with structured messages:

What failed

Why it failed

What to do to fix it

No value in verifier log visualizer.

Anatomy of a Rust Error

```rust
fn main() {
    let s1 = String::from("hello");
    let s2 = s1; // s1 is moved to s2 here
    // Error: s1 is used after move
    println!("{}, world!", s1);
}
```

```
error[E0382]: borrow of moved value: `s1`
 --> src/main.rs:5:28

  |
2 |     let s1 = String::from("hello");
  |         -- move occurs because `s1` has type `String`, which does not
implement the `Copy` trait
3 |     let s2 = s1;

  |              -- value moved here
4 |
5 |     println!("{}, world!", s1);

  |                            ^^ value borrowed here after move
  |
  = note: this error originates in the macro `$crate::format_args_nl`

help: consider cloning the value if the performance cost is acceptable

  |
3 |     let s2 = s1.clone();
  |                ++++++++

For more information about this error, try `rustc --explain E0382`.
```

Why this error is helpful:

- **Error Code**: It provides a specific code (`E0382`) that human can look up for a deep dive into the rules.
- **Visual Labels**: It points to where the variable was **defined**, where it was **moved**, and where it was **used** incorrectly.
- **Actionable Advice**: It often includes a "help" section with a specific code suggestion (like adding `.clone()`) to fix the issue.

Why Rust for BPF?

Agents + Rust = tight loop

- Rust's strictness makes errors apparent earlier
- Rich diagnostics steer the agent in the right direction
- Works in any environments (kernel or firmware)
- C compiles then crashes — agents take longer and sometimes regress to random changes

Previous attempt: Aya

Aya proves Rust-to-BPF is possible but requires strict rules that eliminate most of Rust's advantages:

- No core::fmt, Display, Debug
- No heap — no alloc or collections
- No panic! — no stack unwinding

"C with Rust syntax"

Solution: remove the BPF constraints, not the Rust features.

Verifier Limits That Must Go

These cause agents to give up, treating them as "fundamental limitations"

- 6+ function arguments
- Indirect calls
- Aggregate returns (structs >16 bytes)
- int128 support
- Large stack size
- Deep call chains (>8 frames)
- 1M instruction limit
- 8k jump limit
- Static vs. global function verification gap

Removing these isn't specifically for AI — it widens the scope of what's possible in BPF.
LLMs let us exploit that expanded scope much more easily.

Architecture & Design

Arena-based memory, lifted limits, native Rust features

Design Principles

All program data lives in arena	.data, .bss, heap — one flat address space
Stack limits are lifted	Private stack configurable
Heap works	#[global_allocator] via bpf_arena_alloc/free
ALL Loops verified via widening	Runtime terminated with may_goto
Indirect calls work	PTR_TO_FUNC like subprogs. vtables in .rodata

If Rust code passes compilation, doesn't use unsafe, and sticks to allowed kernel functions — the verifier must accept it.

Memory Model: Arena for Everything

Data type	Section	Access
vtables	`.data.rel.ro`	r/o array map
Constants / strings	`.rodata`	r/o array map
Globals	`.data/.bss`	r/w arena
Heap allocations	`bpf_alloc`	r/w arena

libbpf places most sections into the arena.

vtables contain pointers to functions mixed with r/o data

Why this should work

Every pointer Rust stores is an arena address:

- BTreeMap node → child: arena → arena
- BTreeMap → &str data: arena → .rodata (copy in arena)
- Vec → heap buffer: arena → arena
- Global → BTreeMap root: .bss → arena

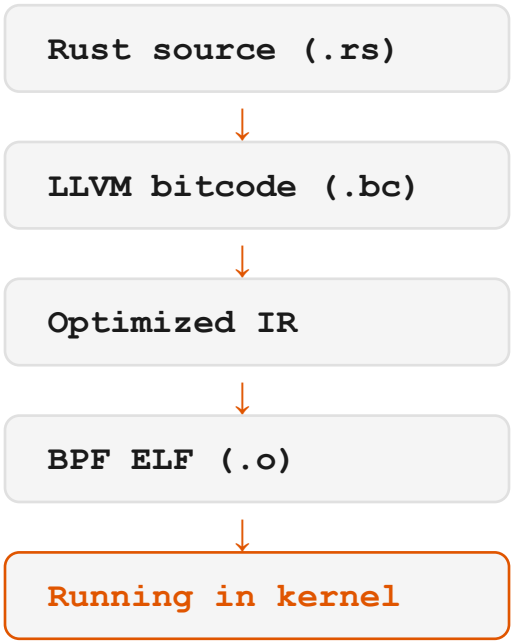
Only restriction: kernel pointers (PTR_TO_FUNC, PTR_TO_BTF_ID) live only in registers, stack and rodata

What will work in Rust-BPF

Rust Feature	Mechanism
`Vec<T>, String, Box<T>`	#[global_allocator] → bpf_arena_alloc
`BTreeMap<K,V>`	Standard alloc collection for non-concurrent execution
`dyn Trait`	PTR_TO_FUNC from vtable loads
`Function pointers`	PTR_TO_FUNC from BPF_PSEUDO_FUNC
`Closures`	PTR_TO_FUNC in `.data.rel.ro`
`for i in 0..n`	Widening at back-edges
`.iter().map().filter()`	LLVM inlines, then widening
`.unwrap(), ?, panic!()`	panic=unwind with landing pads, bpf_throw, formatted message
`Deep call stacks`	8KB+ private stack
`Option<T>, Result<T,E>`	Zero-cost enums
`Generics, monomorphization`	Handled by rustc
`core::fmt / Debug`	Indirect calls + heap

Cannot work: f32/f64 (no FPU) • async/await (may be ?) • std::* (no_std only)

Compilation Pipeline



rustc --target bpfel-unknown-none (obj_is_bitcode, panic=unwind, CO-RE)

llvm-link (merge core + alloc + program)

opt --internalize --globaldce -O2

llc -march=bpf (emit BTF and CO-RE relocs)

libbpf ⟶ arena placement ⟶ verifier ⟶ JIT

No custom tools. No new tools. Upstream rustc, LLVM, libbpf.

Verifier Changes

Necessary changes to unlock Rust on BPF

PTR_TO_FUNC — Indirect Calls

Enables: dyn Trait, function pointers, closures, vtable dispatch

Blocks

- BPF_PSEUDO_FUNC - reused
- BPF_MAP_TYPE_INSN_ARRAY - reused
  ".data.rel.ro" section in frozen bpf array

Challenges

- BTF-based global-prog like may not work
- Full permutations of possible callbacks

```
REJECTED:
r1 += 4              // no arithmetic
*(u64 *)(r1 + 0)     // no deref
*(arena) = r1        // no arena store

ALLOWED:
callx r1             // indirect call
r2 = r1              // register move
if r1 == r2 goto     // comparison
*(r10 - 8) = r1      // stack spill
```

Widening at Back-Edges — Loops

Enables: for i in 0..n, while loops, iterators with runtime bounds

Algorithm

```
At back-edge (jump target ≤ current):
1. Compare current state with loop-head
2. If not converged: widen scalar ranges
   [5,5] ∪ [10,10] → [5,10]
   still diverging?  → [MIN, MAX]
3. Re-verify loop body with widened
state
4. On convergence: continue past loop
```

Rust bounds checks inside loop bodies provide narrowing that makes widened states precise.

Convergence

```
Level 0: concrete   [k, k]
Level 1: range      [a, b]
Level 2: full       [0, U64_MAX]
```

Complexity impact

Without: O(body × iterations) — hits 1M limit

With: O(body × widen_steps) — converges fast

Extended Calling Convention

Functions with >5 arguments

Formatter::pad_integral — 6 registers

```
// Args 1-5: R1-R5 (as today)
// Arg 6:  *(u64 *)(r11 + 8)
// Arg 7:  *(u64 *)(r11 + 16)

// Caller pushes overflow args:
*(u64 *)(r11 - 8) = arg6
*(u64 *)(r11 - 16)  = arg7

// Inside callee R11 = new frame
// caller's (r11-N) = callee's (r11+N)
```

Maps nicely to x86/arm64 calling convention.

Struct return values

Rust returns Result<T,E> where T can be >8 bytes. Matches x86/arm64

```
// sret convention:

// Caller:
alloca %ret_space on stack
R1 = PTR_TO_STACK → %ret_space
R2-R5 = visible args 1-4
call subprog
// result at %ret_space

// Callee:
// R1 is sret pointer
*(u64 *)(r1 + 0) = field_0
*(u64 *)(r1 + 8) = field_1
exit
```

Panic Handling

Panics are not silent. Full error messages via bpf_stream_vprintk before bpf_throw.

```
#[panic_handler]
fn panic(info: &PanicInfo) -> ! {
    use core::fmt::Write;
    let mut stream = BpfStream(0);
    let _ = write!(stream, "{}", info);
    unsafe { bpf_throw(1) }
}
```

Output:

```
panicked at 'index out of bounds:
  the len is 11 but the index is 12',
  alloc/src/.../btree/node.rs:347
```

Unwinding via .bpf_cleanup

1. bpf_throw() called from panic handler
2. Kernel walks BPF call stack
3. For each frame, look up PC in .bpf_cleanup table
4. If match: run cleanup function (Drop impls)
5. If no match: next frame

bpf_throw terminates the program. Unwinder calls drop()s. All arena memory is freed as bulk.

Divide-by-Zero: C vs Rust

Same BPF program, different safety guarantees

C BPF

```
int BPF_STRUCT_OPS(simple_stopping,
                   struct task_struct *p,
                   bool runnable)
{
    if (fifo_sched)
        return 0;

    p->scx.dsq_vtime +=
      (SCX_SLICE_DFL - p->scx.slice)
      * 100 / p->scx.weight;
}
```

Verifier rewrites every division: div-by-zero silently returns 0

Program continues with a wrong value

~5 BPF instructions for the division sequence

Rust BPF

```
bpf_prog!("struct_ops/simple_stopping",
fn simple_stopping(p: *mut task_struct,
                   _runnable: u64) {
    let p = TaskRef(p);
    if FIFO_SCHED.load(Relaxed) {
        return 0;
    }
    let delta =
      (SCX_SLICE_DFL - p.scx_slice())
      * 100 / p.scx_weight() as u64;
    p.set_scx_dsq_vtime(
        p.scx_dsq_vtime() + delta);
});
```

Compiler inserts check: div-by-zero unwinds, calls drop()

~5 BPF insns hot path + panic/fmt code (2000+ insns cold)

Catches bugs instead of silently producing garbage

Example: sched-ext Scheduler in Rust

Port of scx_simple.bpf.c — a global weighted vtime scheduler

```
// Formatted panics
impl core::fmt::Write for BpfStream {
  fn write_str(&mut self, s: &str) ->
    core::fmt::Result {
       let args: [u64; 2] = [s.as_ptr() as u64,
                             s.len() as u64];
       unsafe {
         bpf_stream_vprintk(self.0, b"%.*s\0".as_ptr(),
                            args.as_ptr(), 16)
       };
       Ok(())
  }
}

#[panic_handler]
fn panic(info: &core::panic::PanicInfo) -> ! {
    use core::fmt::Write;
    let mut stream = BpfStream(0);
    let _ = write!(stream, "{}", info);
    unsafe { bpf_throw(1) }
}
```

```
use alloc::collections::BTreeMap;

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

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

}
// #[global_allocator] backed by bpf_alloc
struct BpfAllocator;
unsafe impl GlobalAlloc for BpfAllocator {
    unsafe fn alloc(&self, layout: Layout)
        -> *mut u8 {
        bpf_alloc(layout.size() as u64, 0)
    }
}
```

Standard Rust | alloc collections | kfunc bindings | Formatted panics | unsafe only at edges

Minimal Compiler Changes

rustc target: bpfel-unknown-none

`obj_is_bitcode` `no_std` `panic=unwind`

obj_is_bitcode is key: rustc emits LLVM IR, enabling whole-program LTO.

core + alloc on BPF

After BPF evolution, essentially all of core works except floating point.

`Box` `Vec` `String`

`BTreeMap` `Rc` `VecDeque`

Use alloc::collections::BTreeMap.

rustc frontend + LLVM BPF Backend

- CO-RE
- Support for Rust types in BTF
- Functions with > 5 arguments
- Aggregate returns
- Exceptions and .bpf_cleanup

```
#[btf_relocatable]
#[expect(non_camel_case_types, reason = "Linux kernel type")]
#[repr(C)]
struct task_struct {
    pid: i32,
    tgid: i32,
}
```

Why bpftrace Doesn't Work for Agents

Great for human experts, but agents need guardrails that a DSL can't provide

Out of distribution

bpftrace is a niche DSL. LLMs have seen far less of it in training — they hallucinate syntax, builtins, and probe points that don't exist.

Cryptic errors

Errors are terse with no suggestion. Agents get stuck in retry loops guessing at fixes.

No type system

Dynamically typed. No compile-time signal for wrong struct fields, wrong types, wrong arguments. Mistakes surface at runtime or collect wrong data.

Limited expressiveness

Anything beyond simple aggregations — conditional logic across events, correlating probes, complex data structures — gets awkward. The agent hits a wall.

No incremental iteration

Scripts are standalone one-shots. No way to refactor into functions, add modules, or reuse logic. The agent can't build up complexity step by step.

Agent Toolkit for Live Kernel Debugging

SKILL.md — Kernel Discovery

SKILL.md teaches the agent to:

bpftool btf dump

— explore kernel types, struct layouts

— discover available attach points (tracepoints and kprobe-able functions)

— read function signatures to generate matching Rust probes

— vmlinux.rs with bindings

Agent discovers what to probe and how to attach to it — no guessing

drgn and BPF tracing

drgn for current state:

Inspect live kernel state — walk task lists, read data structures

Agent forms hypothesis from state

BPF for continuous monitoring:

Write a Rust BPF probe to watch the suspect path live — catch the race, the edge case, the intermittent bug

drgn tells agent where to look, BPF watches it happen

Probe crates and sample probes with docs

Common patterns that agent reuses:

Histograms

Stack trace collection

All logic in Rust BPF, no user space post processing, no ringbuf

Agent generates probes from samples and building blocks

Many Challenging Tasks, But Doable.

Unprivileged veristat	Structured verifier errors	Remove artificial limits
PTR_TO_FUNC	Widening for loops	Arena-based memory
Extended calling convention	Panic handling	Native Rust on BPF

BPF evolves to support Rust programs natively. The verifier proves memory safety, not termination. Standard Rust — no BPF-specific annotations.

LLMs improve fast enough. Just wait?