Thoughts on Safety

Kumar Kartikeya Dwivedi

EPFL | Meta

Problem Statement

- BPF provides a baseline safety guarantee of kernel availability.
  - Uphold kernel invariants, and retain the lever to evict misbehaving extensions (either eagerly at load time, or at runtime).
- This is necessary, but not sufficient in practice.
  - Kernel availability is a subset of the availability and progress of the system as a whole.
  - Services on the system can still be negatively impacted even if the kernel recovers.
  - Fuzzy fault isolation: Failures in the extension can propagate to user space applications.

Example 1: XDP

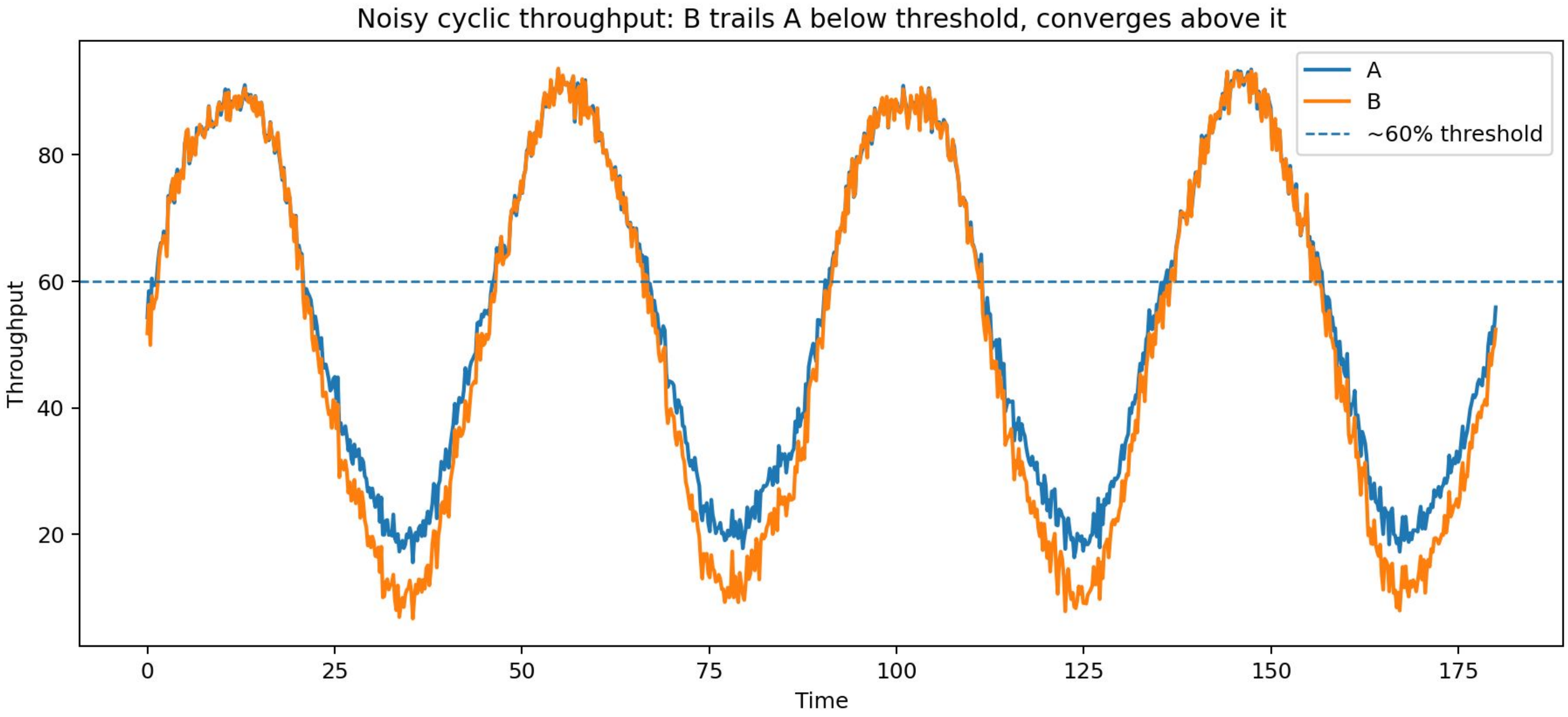
- An XDP load balancer program that accidentally starts dropping traffic due to a logical bug.
- Remote access is blocked; depend on heartbeat logic that evicts the program or do hard reset.
- Mitigations:
  - Rely on code reviews.
  - Ensure testing is representative enough of production traffic to improve probability to hit it.
- Network functions are modular enough (in separating state and logic), such that it is easier to audit their logic, in relative terms.
- Yet, there remains the possibility of such a logical bug slipping through.

Example 2: CPU Scheduling (Starvation)

- The CPU scheduler, due to a logical bug, never schedules a specific task on any CPU.
- The task is stalled; sched-ext has a runtime starvation check, kicks out scheduler in N secs.
- Mitigation:
  - Walk all tasks and check whether they are (eventually) scheduled for a given bound.
  - Do scheduler-side health checks that drain stuck tasks from time to time, before eviction.
- In production, we see 1 or 2 cases every month or so.
  - A small set of cases, that are simply ignored.
  - Some cases pop up on specific topologies, so only show up on rollout on specific hardware.
  - Some are triggered when rolling out to different workloads, which exercise scheduler differently.

Example 3: CPU Scheduling (Performance)

- Work conservation is critical for application performance. Impacts both latency and throughput. Defined as: Ensuring no CPUs go idle when you have (queued) work to do.
- Regression depends on load level of the system, since idle CPUs are only selected below saturation.
- Loopholes:
  - Capacity planning / regression quantification is done at saturation, so we don’t catch this.
  - In production, services run at load levels well short of saturation.
  - We need correlation with product metrics, which doesn’t scale for a scheduling team.
  - So we need extensive testing of all possible combinations
    - (load level x hardware x traffic patterns).


Noisy cyclic throughput: B trails A below threshold, converges above it
A
B
~60% threshold
Throughput
Time
80
60
40
20
0
25
50
75
100
125
150
175

Observations

- Clearly a need to have more confidence in the behavior of kernel extensions.
- Compared to other software, impact of bugs in BPF programs is magnified due to the control they can exert over system behavior, and services on the system.
- Behavior of the extension is not isolated to the in-kernel program, but also affected by the associated user space application and its functional logic.
  - E.g., sched-ext schedulers are composed as a user space component and BPF program.
- Testing is a mandatory line of defense, but coverage is difficult over a huge state space.

Designing Kernel Interfaces

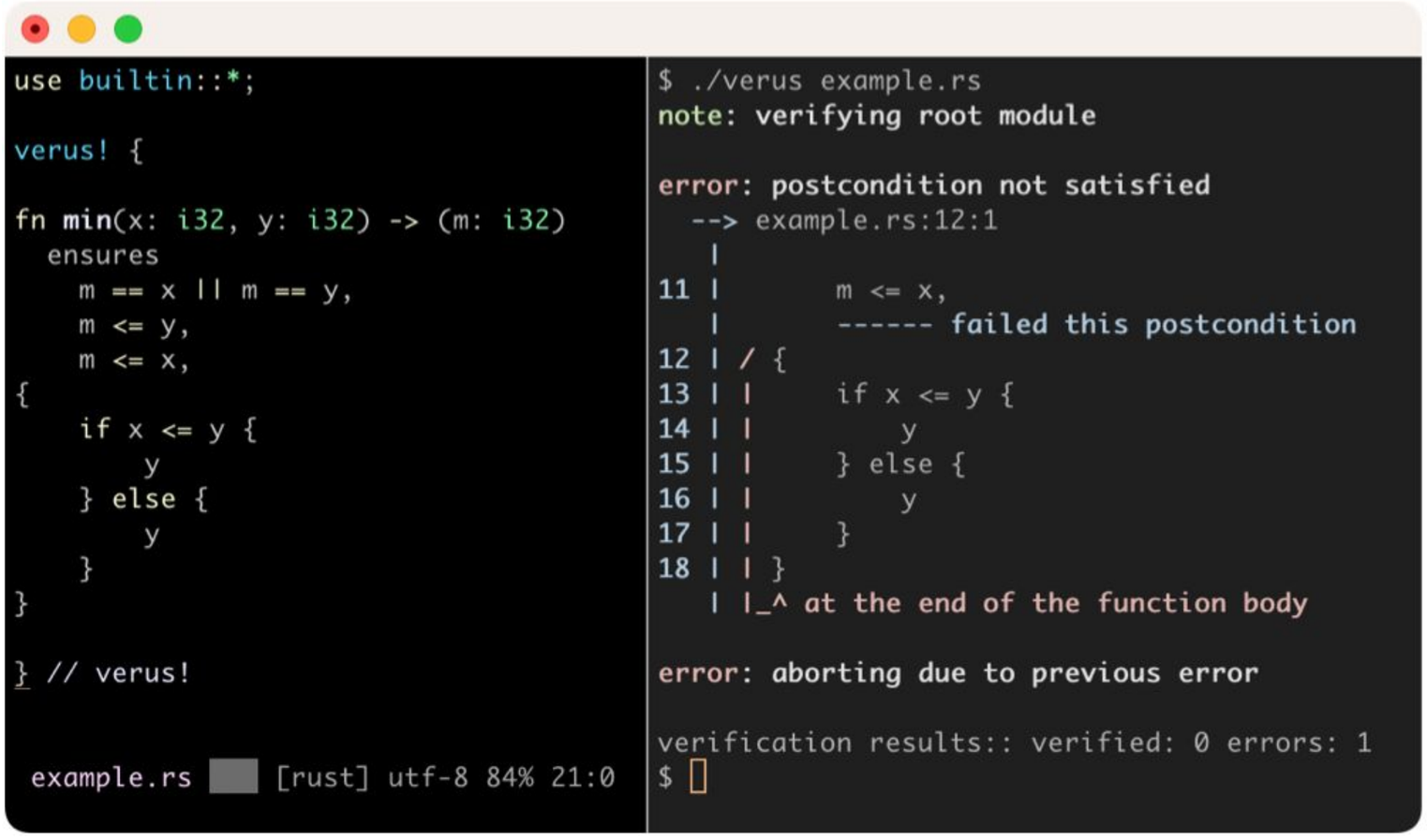
- Social problem: People who design kernel interfaces are not necessarily verification experts.
- They understand their subsystem’s core invariants, and use BPF’s type system to enforce it.
- Stronger guarantees requires precise modelling of the program’s behavior.
- We increasingly treat extension’s private state in arenas as a black box, making most complex properties that require reasoning about extension’s memory state undecidable.
- The line on what is needed for strict correctness of extensions is fuzzy / context-dependent.
  - Work conservation is desirable in general, but some use cases may eschew it.
- Being overly eager and precise about defining correctness hurts exploration and innovation.

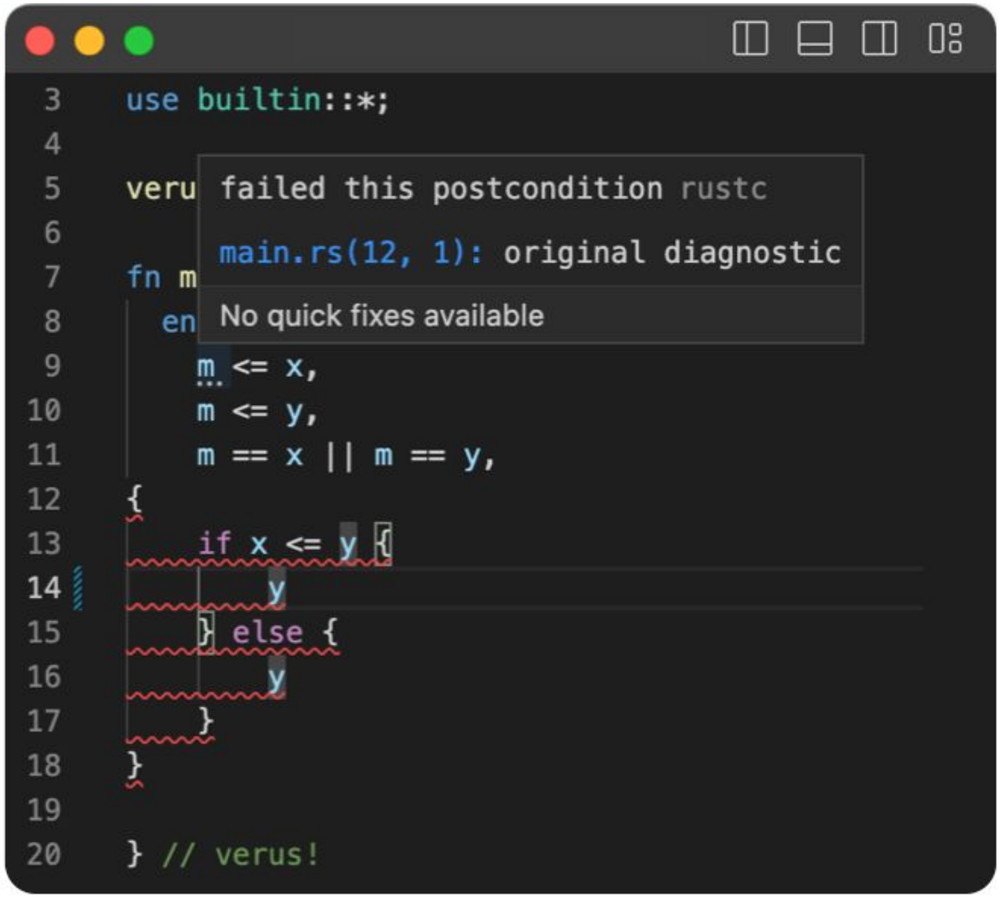
Mistakes

- bpf_obj_new() based data-structures (bpf_list, bpf_rbtree).
- Inherited and coupled constraints of kernel object lifetime and safe mutation into the program's own data structure management.
- The verifier would severely restrict usage to ensure object ownership can be tracked statically so as to release individual object memory back to kernel.
- Consequently, a lot of algorithms and patterns became inexpressible.
- We don't care about individual object lifetimes of memory under the sole purview of the program, but we didn't do separation of concerns.

Rust-BPF + Verus

- The invariants of the kernel interface (helpers, kernel object access, etc.) are enforced by the verifier.
- The remainder of program behavior is abstracted into arena access and runtime-checked loops, relaxing programming constraints.
- Layer Rust on top of arenas to claw back memory safety for the program’s own resources.
- Better than writing programs in C, but still not enough for cases we described earlier.
- We can layer Verus on top, which is an automated verification toolchain built on top of Rust.


use builtin::*;
verus! {
fn min(x: i32, y: i32) -> (m: i32)
ensures
m == x || m == y,
m <= y,
m <= x,
{
if x <= y {
y
} else {
y
}
}
} // verus!
example.rs [rust] utf-8 84% 21:0
$ ./verus example.rs
note: verifying root module
error: postcondition not satisfied
--> example.rs:12:1
11 | m <= x,
| ------ failed this postcondition
12 | / {
13 | | if x <= y {
14 | | y
15 | | } else {
16 | | y
17 | | }
18 | | }
| |_^ at the end of the function body
error: aborting due to previous error
verification results:: verified: 0 errors: 1
$


use builtin::*;
failed this postcondition rustc
main.rs(12, 1): original diagnostic
No quick fixes available
m <= x,
m <= y,
m == x || m == y,
{
if x <= y {
y
} else {
y
}
}
} // verus!

Practical Observations

- We do not need the same level of safety and correctness at every point of development lifecycle.
  - When I am exploring a scheduler optimization, I don’t want to pay the cost of correctness and specify guarantees upfront.
- Different users of the same extension interface (e.g., CPU schedulers, XDP) may have different tolerance for correctness, and different needs for flexibility.
- TL;DR: It is unlikely one solution works for everyone.

Requirements

- Strength of safety requirements for a given user fall on a spectrum.
  - Depends on their production environment idiosyncrasies, the development lifecycle they are in (exploring vs productionizing), etc.
- The kernel verifier provides worst-case bounds in case of misbehavior.
- The rest of safety properties are stacked on top of the kernel verifier, successively trading flexibility for provable and decidable correctness.

Example: Work Conservation

- Work conservation is a global system property at an instantaneous time.
  - If there exists any overloaded CPU, then for every core C', core C' is not idle.
- Ipanema: Restricted DSL for expressing (complex) scheduler policies that are verifiable (wrt CWC) from Inria, Paris. Folks prove Concurrent Work Conservation, but that is out of scope.
- Core idea: Restrict arbitrary mutation of queues, tracking of load manually, uncontrolled reads of concurrent remote CPU state, etc. The policy is limited to small narrow interface.

Provable Multicore Schedulers with Ipanema: Application to Work Conservation

Baptiste Lepers
baptiste.lepers@sydney.edu.au
University of Sydney

Redha Gouicem
Damien Carver
first.last@lip6.fr
Sorbonne Université, LIP6, Inria

Jean-Pierre Lozi
jean-pierre.lozi@oracle.com
Oracle Labs

Step 1: Invariants and Proof Obligation

- Each scheduler helper (in the new internal interface, on top of kernel interface) has:
  - Precondition:
    - What must be true before the operation.
  - Postcondition:
    - What the operation guarantees afterward.
  - Frame condition:
    - What the operation cannot change (which we rely on for composing the proof).
  - Abstract State and Valid State Transitions:
    - State maintained by the scheduler used for decision making, and state transitions.

Step 2: Individual Proofs

- Please see the paper for details; We discuss a small (incomplete) example for building intuition.
- Consider abstract state:
  - nr_tasks[cpu] == Runnable tasks on a given CPU ‘cpu’.
  - idle(cpu) := nr_tasks[cpu] = 0
  - overloaded(cpu) := nr_tasks[cpu] > 1
- Rule out:
  - overloaded(A) ^ idle(B)
- Strategy: Restrict manipulation of queues arbitrarily.
  - Let’s only allow push(task, cpu), and pop(cpu).

Step 2 (a): Push

- push(task, cpu)
  - Require:
    - Task is runnable.
    - CPU is eligible.
  - Ensure:
    - nr_tasks[cpu]++

Step 2 (b): Pop

- pop(cpu)
  - Require:
    - Task is runnable.
    - CPU is eligible.
  - Ensure:
    - nr_tasks[cpu]--

Step 2 (c): WC Proof

Case 1: push(task, cpu)

- push can only increase nr_tasks[cpu]
- If cpu does not become overloaded.
  - No new bad state is possible.
- If cpu becomes overloaded:
  - push must prove there is no idle eligible cpu.
  - If there is, place the task on one.

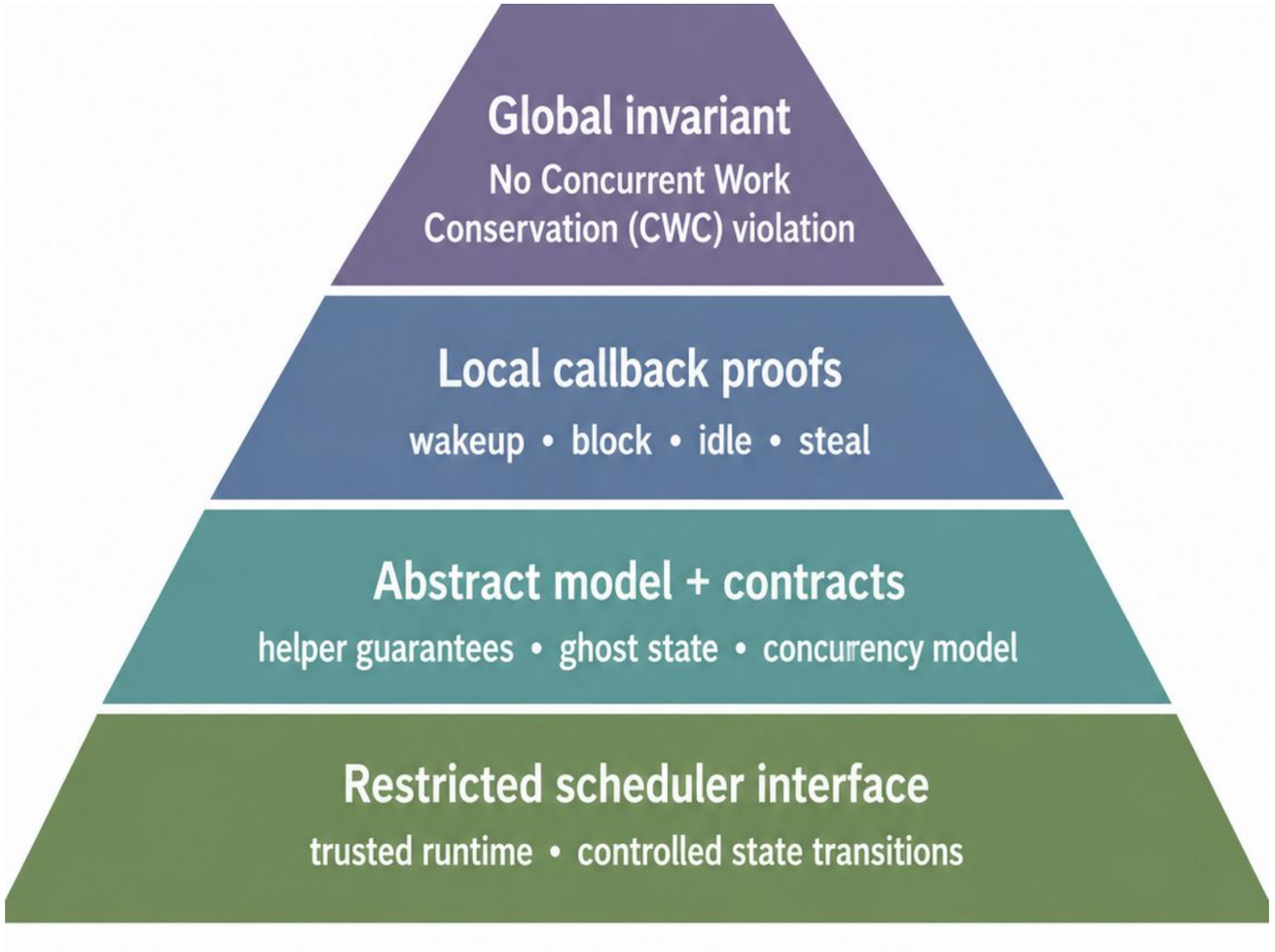
Step 2 (d): WC Proof

Case 2: pop(cpu)

- pop can only decrease nr_tasks[cpu]
- If cpu does not become idle:
  - no new bad state is possible.
- If cpu becomes idle:
  - pop must first try to steal from an overloaded cpu.
  - pop may return empty only after proving:
    - ∄ cpu. overloaded(cpu)

Step 3: Composing the Final Proof

- Therefore:
  - push cannot create unexplained overload.
  - pop cannot create unexplained idleness.
  - All scheduler callbacks are compositions of push/pop
    - ⇒ No callback can introduce the first bad state. overloaded(A) ^ idle(B)
    - Q.E.D.


Global invariant
No Concurrent Work
Conservation (CWC) violation
Local callback proofs
wakeup • block • idle • steal
Abstract model + contracts
helper guarantees • ghost state • concurrency model
Restricted scheduler interface
trusted runtime • controlled state transitions

Summary

- Various functional properties of extensions impact system availability and performance.
- Some of the overall performance and reliability properties are as critical as kernel not crashing.
- Testing is complementary to improved verification over the input space of the program.
- We need to build more constrained interfaces and frameworks on top of the kernel's interface to trade flexibility for tractable static verification of correctness properties.
- Ensuring that BPF's evolution and direction fosters such an ecosystem and allows experimentation with various approaches on top to strengthen guarantees.

Acked-by: Eduard Zingerman <eddyz87@gmail.com>
Reviewed-by: Emil Tsalapatis <emil@etsalapatis.com>

Meta

Motivation

- BPF provides a baseline level of safety, which is necessary but not sufficient.
- System availability, or other desirable (performance) properties aren’t always defined by crash safety.
  - Easier case: XDP program should never unconditionally start returning XDP_DROP due to programming errors. Relatively easier to audit, and surface during testing.
  - Harder case: Ensure my CPU scheduler doesn’t starve critical threads. Harder to audit statically (with the current interface), but doable and hopefully surfaces in testing.
  - Pipe dream: Tie my performance objectives to invariants enforced by the scheduler, e.g. fairness, or not violating work conservation. Harder to detect, harder to test.

Challenges

- Additional constraints to verify richer semantics limits implementation and innovation freedom.
  - E.g. ignoring work conservation is sometimes what we want, so enforcing that in the extension framework would limit possibly implementations.
- Requirement for safety is also dependent on the development cycle.
  - When I am exploring a hypothesis, I can trade complete correctness to be able to move faster and verify my hypothesis.
    - The next step of adding more verification steps increases my confidence, but it needs to be incremental.

– Social challenges

- Kernel developers are not automated verification experts, esp. those who design the kernel interface.
- Aim for maximum freedom and availability in worst-case situations.

Solution

- Split the components of the end-to-end solution and let experts at each level own the individual component.
- Design things in a way that these individual components allow stacking safety properties together, such that we can climb up the strengthening pyramid.

The kernel cares about kernel availability – a core invariant that once an extension is attached, we retain the ability to restore the host kernel back to its prior steady state upon removal, and retain the lever for removal.

Not perfect, e.g. XDP can lock out the person, but you could still imagine a host-level probe / health check that retains the ability to unload the program. So it is possible to argue that the lever still exists. Likewise for sched-ext.

Solution

Kernel extension framework also tries to establish separation between constraints imposed for the satisfaction of the kernel interface (callbacks, helpers, resource access) from the behavior specific to the program (its own extension data, time spent on the CPU, etc.)

- We’re already moving in this direction with arenas, which is great.

As such, we can define a richer interface for writing the program on top of the kernel interface which is defined to ensure the kernel’s invariants are upheld, and that we don’t end up in a irrecoverable state.

Intuition

Rust-BPF is sort of already doing that in some sense.

- C BPF programs are not memory safe by construction for their own arena.
- Rust BPF however is, unless it breaks that guarantee with the unsafe escape hatch.

- The properties Rust BPF ensures is a subset of the total functional correctness envelope.