DPDK SUMMIT

Powering the Future of Networking Software

12–13 May 2026

AI-Assisted Formal Verification of the DPDK eBPF Verifier

Marat Khalili

Claudia Cauli

About Us

Marat Khalili
Principal Software Engineer

Claudia Cauli
Principal Research Engineer and Formal Methods Team Lead

Cloud Reliability Lab
Huawei Ireland Research Center

About eBPF

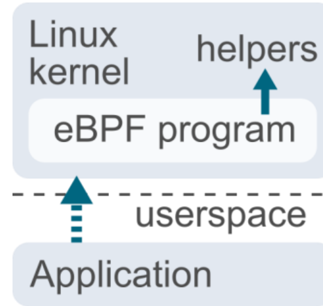
Multi-billion dollar ecosystem, does not really require introduction.

Some use cases, common to both Linux and DPDK:

- **Networking** (forwarding, filtering & transforming packets)
- **Security & Observability** (auditing and enforcing network and system access)
- **Cloud Cost Optimization & FinOps** (controlling resources utilization)
- **Zero-Code-Change Production Debugging** (*when push comes to shove*)
- and many more…

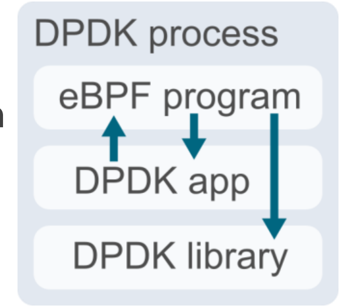
Linux eBPF vs DPDK eBPF Architectural Differences

Linux eBPF



- Executed within the kernel at predefined hook points.
- Any interactions with the apps need to pass through the user-kernel boundary.
- Helper functions are either maintained as part of the Linux tree or need to be loaded in loadable kernel modules (LKM).
- Useful for monitoring or changing Linux kernel behavior, including managing traffic through kernel-controlled interfaces.

DPDK eBPF



- After load can be executed within the DPDK application as any other C function.
- Executes and calls back at native function speeds.
- Can be given direct access to any other functions or data structures within the DPDK application or libraries (or even drivers).
- Useful for monitoring or changing behavior of DPDK-based applications and managing traffic through DPDK-controlled interfaces.

TLDR: DPDK provides much more flexibility in how eBPF is used and what it can do.

DPDK SUMMIT
Powering the Future of Networking Software

Example Use Scenarios

Situation	Solution
Malformed or excessive traffic is crashing or DDoS-ing our service, we need to redirect or filter out very specific packets not easily selectable by available rules.	Safe eBPF-based filter with arbitrary logic is developed and installed by AI or human in minutes without restarting the application or losing any payload data.
We suddenly realized we need to collect telemetry data from the diverse park of DPDK-based applications, a thought that completely missed our plans during their development and deployment.	"Magic OBD2 port" in the shape of eBPF lets us add and maintain plugins collecting any necessary internal data and sending it over the network, all without touching the core application logic and risking regressions.

In both cases safety of the eBPF programs plays crucial role!

But we need to make sure this safety is actually guaranteed.

eBPF Safety and Verification

eBPF Verifier (Validator)

- Crucial component verifying safety of eBPF programs on load (not during the execution).
- Performs multiple checks, such as validating instruction opcodes, checking execution graph for loops, dangling branches, dead code etc.
- One of the most important parts is evaluation of possible values of local variables along all code paths to guarantee memory safety of all memory accesses and helper function calls.
- Allowed to have false positives (reject corner-case valid programs) but must never have false negatives (accept dangerous programs).

eBPF Safety Guarantees

- Only dereferences or passes to the helper functions pointers to memory areas it was explicitly given access to, preventing application crashes.
- Never reads or returns uninitialized data, preventing data leaks.
- Has no (unbounded) loops, always returns within a limited number of instructions, preventing freezes and latency spikes.
- No undefined behavior!

To Verify the Verifier

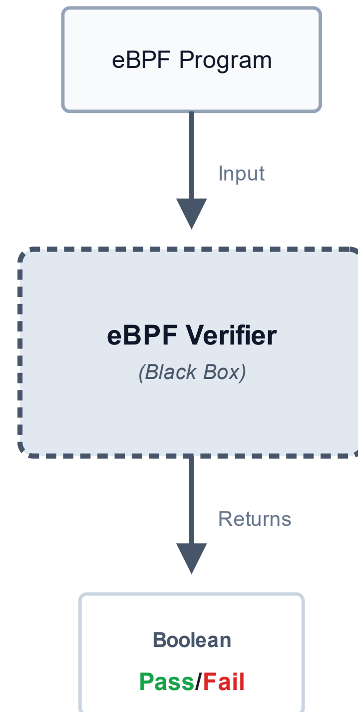
```
testpmd> bpf-load rx 0 0 M examples/bpf/t1.o
BPF: evaluate(): mbuf access with variable offset at pc: 17
-22:Invalid argument
```

Verifier is a *Black Box*: for any program it only returns one bit of data: Pass or Fail.

Human-readable diagnostic message may tell us type of error, but not how we ended up there.

Problems:

- Debugging validation failures when writing eBPF program.
- Writing tests to make sure verifier rejects all invalid programs.

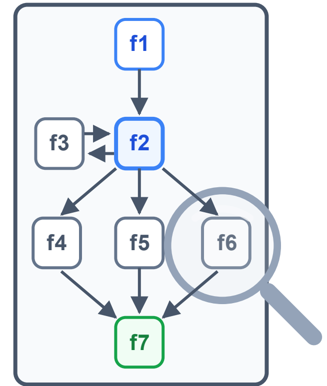


Idea: Unit Tests for Specific Implementation Functions

Possible solution: access to individual functions and data structures in `bpf_validate.c` and write tests for the expected behavior and contents.

- Nice to have, but time-consuming to write and keep updated.
- Coupled to specific implementation and have to change with it.
- Not a replacement for feature-tests of user-visible behavior.
- Does not help user debug validation of their eBPF program.

Can we do better?



Idea: gdb-like Interface for eBPF Verifier

Q: How can we get a little more insight? What is the minimum we can assume about the eBPF program evaluation by verifier regardless of the implementation?

A: Verifier tracks possible variable values for each instruction along each code path and can answer questions about allowed memory accesses and possible jumps.

Let's build an API for this!

```
BPF: evaluate(): mbuf access with variable
offset at pc: 17
Catchpoint 1 on validation-failure.
Validation failed.
=>          17:  ldxh r1, [r1 + 2]

(validate) info r1
   r1:  %mbuf<128, 2176> + 14..77

(validate) list program
            16:  add r1, r2
=>          17:  ldxh r1, [r1 + 2]

(validate) info r2
   r2:  0..63

(validate) may r2 == 0
YES

(validate) may r2 s>= 64
NO
```

DPDK SUMMIT
Powering the Future of Networking Software

Using New API To Write Tests (and catch real bugs)

```c
int check_post(struct rte_bpf_validate_debug *debug) {
    assert(rte_bpf_validate_debug_may_jump(debug,
         &(struct ebpf_insn){
            .code = (BPF_JMP | EBPF_JEQ | BPF_K),
            .dst_reg = BPF_REG_2,
            .imm = 2,
        }
    ));
}
int test(void) {
    struct rte_bpf_validate_debug *debug =
        rte_bpf_validate_debug_create();
    rte_bpf_validate_debug_break(debug, 4, check_post);
    const struct rte_bpf_prm prm = {..., .debug = debug};
    struct rte_bpf *bpf = rte_bpf_load(&prm);
    /* cleanup and return... */
}
```

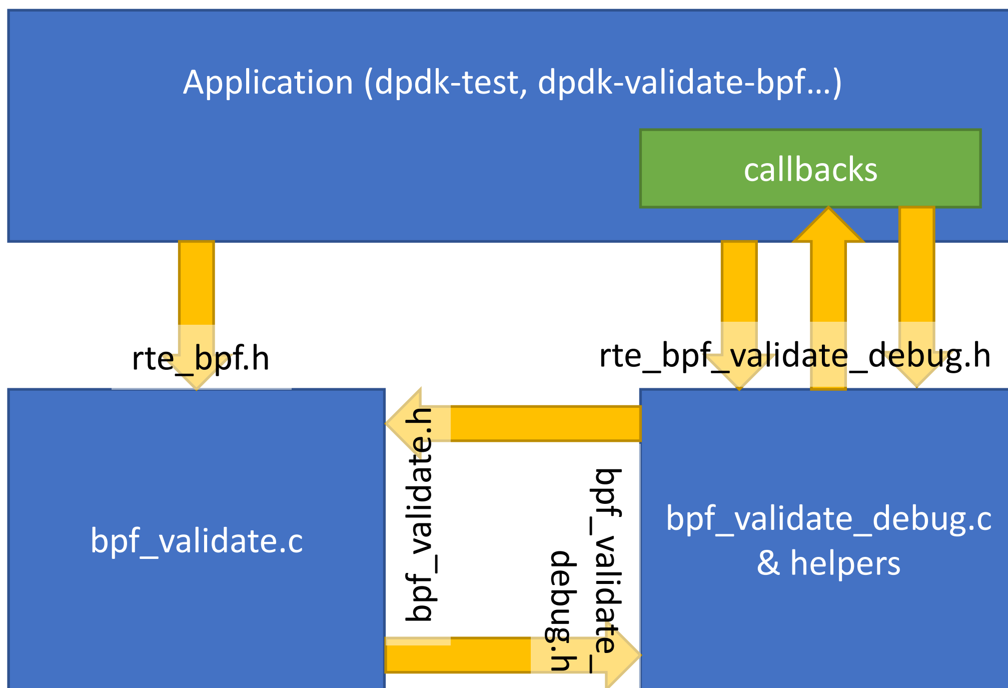
```
; initialize return value
0: mov r0, #0x0
; load unknown value from memory
1: ldxdw r2, [r1 + 0]
; jump to exit if positive
2: jsgt r2, #0x0, L5
; tested instruction: negate value
3: neg r2
; break here and check the state
4: mov r0, #0x1
; make sure our program ends
5: exit
```

(Negation of a negative can produce any positive value, for example 2.)

DPDK SUMMIT
Powering the Future of Networking Software

Architecture of the Solution

- The debug feature is structured as a subcomponent within `lib/bpf`, changes to the main validation code are intentionally reduced to minimum.
- Validation subcomponent interacts both ways with the debug subcomponent via two new internal headers.
- Header `rte_bpf_validate_debug.h` declares all new external API.
- To initiate debugging user only needs to create debug instance, register callbacks on it, and pass it during BPF load.
- The rest happens within the callbacks.



DPDK SUMMIT
Powering the Future of Networking Software

The Problem of Scale

We found one bug. How do we find all of them?

- 100s of eBPF instruction opcodes…
- 10s of corner cases for each of them…

Many 1000s of tests needed with precise math for each…

Verifying the Verifier, Formally

From one bug to *all*

- **Huge state space** (explodes over 100s of opcodes)
- **Tests can only sample** (even property-based tests)
- **One missed input is an exploit**
- We need to ***prove*** the invariants
- We need ***guarantees***, not only confidence.

What **correctness** means for the eBPF Verifier

Soundness

- Accepted → Valid
- The eBPF Verifier ***never accepts an invalid program***.
- **Must-have**
- FN = **Exploit**

Completeness

- Valid → Accepted
- The eBPF Verifier ***accepts every valid program***.
- **Nice-to-have**
- FP = **Conservative rejection**

Soundness first

One property – 15+ bugs

Comprehensive soundness needs:

- **Termination**
- **Type safety**
- **Memory safety**
- **Definite initialization**
- **Control flow safety**
- **Pointer subtyping**
- *…and more*

Range Invariant Preservation:

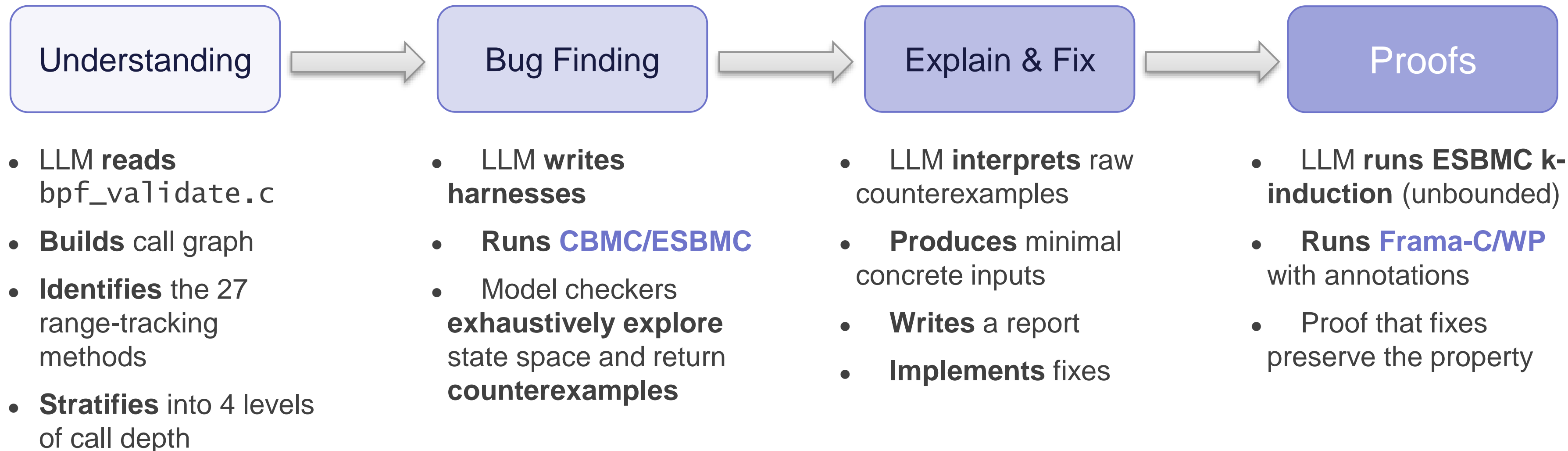
Every range-tracking operation must preserve the invariant

```
s.min <= s.max && u.min <= u.max
```

Foundational. Scoped. Provable

DPDK SUMMIT
Powering the Future of Networking Software

AI-Assisted Formal Verification – The Setup



Three open source tools. One AI agent.

~3 hours to first results.

Bug: Signed Overflow in `eval_divmod()`

```
static const char *
eval_divmod(uint32_t op, struct bpf_reg_val *rd, struct
bpf_reg_val *rs, ...)
{
    /* ... unsigned logic ... */
    /* signed logic */
    if (rd->s.min == rd->s.max && rs->s.min == rs->s.max) {
        if (rs->s.max == 0) return "division by 0";
        if (op == BPF_DIV) {
            rd->s.min /= rs->s.min;
            rd->s.max /= rs->s.max;
        } else
        /* ... MOD case ... */
    /* ... */
```

LLM → **ESBMC counterexample**:

```
rd.s.min = INT64_MIN
rd.s.max = INT64_MIN
rs.s.min = -1
rs.s.max = -1
```

`INT64_MIN / -1` UB in C

DPDK SUMMIT
Powering the Future of Networking Software

Bug: Signed Overflow in `eval_divmod()`

```
static const char *
eval_divmod(uint32_t op, struct bpf_reg_val *rd, struct
bpf_reg_val *rs, ...)
{
    /* ... unsigned logic ... */
    /* signed logic */
    if (rd->s.min == rd->s.max && rs->s.min == rs->s.max) {
        if (rs->s.max == 0) return "division by 0";
        if (op == BPF_DIV
            && rd->s.min == INT64_MIN && rs->s.min == -1)  {
            eval_smax_bound(rd, msk);
        else if (op == BPF_DIV) {
            rd->s.min /= rs->s.min;
            rd->s.max /= rs->s.max;
        }
        /* ... MOD case ... */
```

LLM's Fix Bespoke overflow check & widen to top

LLM's Suggestion *"or compute using unsigned arithmetic"*

DPDK SUMMIT
Powering the Future of Networking Software

Bug: Signed Overflow in `eval_divmod()` – Twist!

```
static const char *
eval_divmod(uint32_t op, struct bpf_reg_val *rd, struct
bpf_reg_val *rs, ...)
{
    /* ... unsigned logic ... */
    /* signed logic */
    if (rd->s.min == rd->s.max && rs->s.min == rs->s.max) {
        if (rs->s.max == 0) return "division by 0";
        if (op == BPF_DIV) {
            rd->s.min /= rs->s.min;
            rd->s.max /= rs->s.max;
        } else
        /* ... MOD case ... */
    /* ... */
```

DPDK does not implement SDIV

SDIV is in eBPF ISA

In DPDK, DIV is *always unsigned* at bytecode level

DPDK SUMMIT
Powering the Future of Networking Software

Bug: Signed Overflow in `eval_divmod()` – Twist!

LLM + ESBMC found a real bug

But

The validator's abstract interpretation **did not match the runtime semantics**.

We only asked the LLM to verify *Range Invariant Preservation*.

→ This is a cross-layer property:
"abstract domain agrees with runtime semantics"

DPDK SUMMIT
Powering the Future of Networking Software

No single tool finishes the job

CBMC	ESBMC	Frama-C/WP
Bounded Model Checking	Bounded Model Checking k-induction	Deductive Verification
SAT-based	SMT-based	Algebraic, ACSL annotations
Bug-finding	Bug-finding + unbounded proof	Method-level proofs
Timed out on bitvector mul/div	Timed out on bitvector mul/div	Converges on eval_mul, divmod, etc.

Bug-finding ≠ Proof. Different jobs. Different tools

What's next – and where you come in

- **What we did**

First concrete step towards verifying the verifier.
Verified **one property**, Range Invariant Preservation.
15+ bugs, 27 functions.

- **What's next**

Termination · mem safety · type safety · pointer subtyping · … · **soundness**.

- **Takeaway**

AI-assisted Formal Verification is **now in reach for all of us**

Where next within DPDK should we point?

DPDK SUMMIT
Powering the Future of Networking Software

Supplementary Slides

DPDK eBPF Validator Fixes (first batch*)

Commit	Fixed Problem
fix BPF_LDX | EBPF_DW signed range	Signed range was initialized incorrectly.
fix EBPF_JSLT | BPF_X evaluation	Instruction evaluation was missing.
fix BPF_NEG of INT64_MIN and 0	Corner cases were not handled properly.
fix BPF_DIV and BPF_MOD signed part	Calculated signed division instead of unsigned, crashed with SIGFPE.
fix BPF_MUL ranges minimum typo	Used data for the wrong register.
fix BPF_MUL signed overflow UB	Large ranges triggered undefined behavior.
fix BPF_JGT/EBPF_JSGT no-jump max	Used wrong end of the range.
fix BPF_JMP source range calculation	Only updated state of one register out of two.
fix BPF_JMP empty range handling	Tried to evaluate unreachable code path running into all kinds of troubles.
fix BPF_AND mins calculation	Used incorrect math formula.
fix BPF_LSH shift-out-of-bounds UB	Large shifts triggered undefined behavior.
fix BPF_OR unsigned min calculation	Used incorrect math formula.
fix BPF_OR signed min calculation	Handled negative numbers incorrectly.
fix BPF_SUB signed max zero case	Overflow detection used wrong ends of the range.
fix BPF_XOR signed min calculation	Handled negative numbers incorrectly.

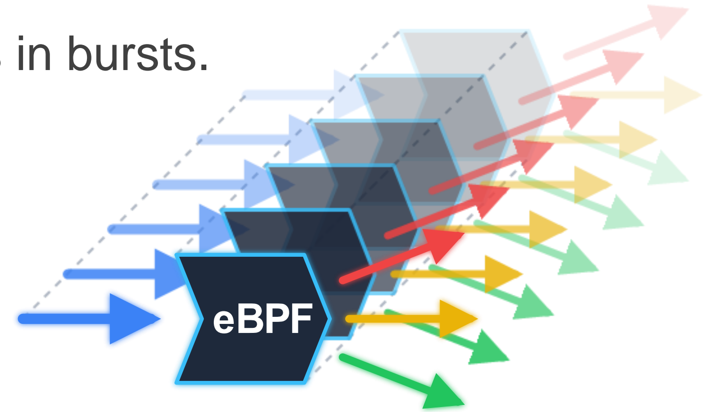
* more under way

Roadmap and Ideas for DPDK/eBPF Future Development

- **eBPF “maps” (container library)**
- DPDK eBPF SDK (simplify linking against DPDK functions) and helpers library
- Multiple program arguments (for seamless eBPF-based callbacks)
- New types: atomics, spinlocks, timers, per-thread/per-lcore variables…
- BPF Type Format (BTF)!
- Better Linux compatibility (XDP support?)
- Verifier improvements:
  - NULL pointers (interoperability with DPDK functions)
  - Track memory areas (reduce false negatives, improve safety)
  - Track dependent variables (reduce false positives for complex programs)

DPDK SUMMIT
Powering the Future of Networking Software

SIMD & Burst Handling: Outrunning Native C Code



Part of DPDK speed comes from handling packets in bursts.

Introducing SIMD JIT eBPF compiler idea:

- Single AVX-512 instruction can operate on 8 eBPF registers at once.
- We already build and analyze eBPF program Control Flow Graph (CFG) as part of verification.
- CFG edges only require straightforward changes to JIT instructions issued. CFG nodes add reconvergence points for maintaining lanes utilization.
- Need SIMD versions of external functions, but can fallback to sequential calls.
- No change to eBPF source code is needed!

External eBPF Libraries: Just Use `bpftime/rbpf/ubpf`?

Pros of external runtimes

- *Possibly* more complete, support more architectures, faster
- *Possibly* better tested (?)
- Someone else's headache!

Cons of external runtimes

- Built with C++ or Rust
- Bring additional dependencies
- Versions compatibility problem
- Do not account for DPDK specifics

Verdict

It's not the code that we write, it's the interfaces we develop along the way.

DPDK needs own runtime and interoperability with external ones.

DPDK SUMMIT
Powering the Future of Networking Software

DPDK

SUMMIT

Powering the Future of Networking Software