A Mechanized Semantics For eBPF And Its Applications In Program Verification

Marcos Geraldo Braga Emiliano
Programa de Pós-Graduação em Ciência da Computação,
Universidade Federal de Ouro Preto
Ouro Preto, Brazil
marcos.emiliano@aluno.ufop.edu.br

Rodrigo Ribeiro
Programa de Pós-Graduação em Ciência da Computação,
Universidade Federal de Ouro Preto
Ouro Preto, Brazil
rodrigo.ribeiro@ufop.edu.br

Abstract

The absence of a formally verified semantics for eBPF hampers the development of verified analysis tools and leaves JIT compilers and static analyzers without an authoritative reference. This paper presents a mechanized formalization of eBPF in Lean 4, comprising: (i) a small-step operational semantics over 64-bit machine words and flat byte-addressed memory; (ii) a separation logic whose assertions are predicates over register files and partial heaps; (iii) a fully proved sound frame rule, with a semantic soundness theorem connecting the syntactic derivation system to the operational model; and (iv) a machine-checked, end-to-end correctness proof for an IPv4/TCP packet-filter program.

Keywords

eBPF, Separation Logic, Hoare Logic, Operational Semantics, Lean 4.

1 Introduction

eBPF is a programmable virtual machine embedded in the Linux kernel that enables user-defined programs to execute safely inside the kernel without modifying kernel code or loading a module. Originally conceived as a packet-filter mechanism [12], it has grown into a general-purpose kernel extension platform spanning three domains: *networking*, including load balancing [6], XDP-based packet processing [24], and firewalling [3, 5]; *security*, through syscall filtering and runtime policy enforcement; and *observability*, including low-overhead kernel tracing and profiling [18]. The *kernel verifier* enforces safety by rejecting programs before loading, checking for out-of-bounds memory accesses, uninitialized registers, and unbounded loops.

Despite this wide adoption, the normative specification of eBPF is an informal prose document [19, 21, 22], supplemented only by the kernel source. This informality has real consequences: independent JIT compilers for different CPU architectures have diverged in their treatment of edge cases, and the kernel verifier has contained soundness bugs [14]. An active conformance test suite [10] documents these divergences empirically, but test suites cannot certify all programs. A machine-checked formal semantics provides an authoritative reference against which verifiers, JIT compilers, and static analyzers can all be validated.

Prior formalization efforts have addressed narrower goals. Nelson et al. [14] verify BPF JIT correctness using Serval but do not mechanize a program-level semantics or separation logic. Yuan et al. [25, 26] build a verified eBPF interpreter in Rocq for IoT scenarios, targeting simplified runtime models without a separation logic. No prior work provides a mechanized separation logic for eBPF with a proved-sound frame rule, which is the foundational tool for modular, compositional reasoning about memory safety.

This paper presents a mechanized formalization of eBPF in Lean 4 [13]. More specifically, the main contributions are:

- *Small-step operational semantics* covering the core eBPF ISA (ALU64/32, loads, stores, conditional and unconditional jumps, helper calls, LDDW, and atomics), where multi-step derivations live in `Type` to support the call-safety analysis of the soundness proof.
- *Separation logic* with assertions over register files and partial heaps, a separating conjunction, weakest-precondition transformers for each instruction class, and `HeapOnly` frame assertions that remain valid across register-modifying ALU steps.
- *Soundness theorem* (`htriple_sound`): every `HTriple` derivation implies the corresponding `SemTriple`, a statement universally quantified over all terminating call-safe runs that explicitly carries a silent frame heap, making the frame rule an intrinsic property of the semantic triple.
- *End-to-end verified packet filter*: an IPv4/TCP packet-filter program verified using direct `HTriple` derivations and lifted to semantic correctness via `htriple_sound`. The development also includes a fuel-based and a fuel-free interpreter with bridge theorems connecting both to `SemTriple`; these results are available in the repository [2] and omitted here for space.

We describe all results using LaTeX notation; all mathematical definitions follow their Lean implementation as closely as possible. The full development is available at [2].

The rest of this paper is organized as follows: Section 2 reviews background on eBPF and separation logic. Sections 3–5 present the semantics, separation logic, and soundness theorem. Section 6 presents the case studies. Section 7 surveys related work and Section 8 concludes.

2 Background

2.1 An Overview Of eBPF

The eBPF VM was originally designed for packet filtering and has grown into a general-purpose kernel extension platform. The VM [12, 24] provides eleven 64-bit registers (`r0`–`r9` plus a read-only frame pointer `r10`), flat byte-addressed memory, and an implicit program counter. Programs are bytecode arrays loaded from user space; the kernel JIT-compiles them to native code after verification. Our formalization covers the core instruction set: ALU64/32 (arithmetic, logic, shifts, byte-swap), memory loads and stores (byte

through dword), conditional and unconditional jumps, helper calls, wide-immediate loads (LDDW), and atomic read-modify-write operations. *Helper calls* are kernel-provided functions that eBPF programs invoke to access kernel services unavailable as plain bytecode, such as reading packet data, querying maps, or obtaining timestamps.

The registers follow a fixed *calling convention*. Register `r0` holds the return value of both helper calls and the program itself. Registers `r1`–`r5` carry arguments: at program entry, `r1` points to the *context object* (e.g., a packet buffer or socket metadata) passed by the kernel. Registers `r6`–`r10` are *callee-saved*: their values are preserved across helper calls, allowing the program to store long-lived data there without spilling to the stack. Register `r10` is a read-only frame pointer to the 512-byte BPF stack, which is private to the program and not shared with helpers.

eBPF programs interact with the kernel through *helper calls*. Each helper is identified by a 32-bit function ID; the kernel exposes a fixed set of helpers (e.g., `bpf_map_lookup_elem`, `bpf_skb_load_bytes`) that programs may invoke. Arguments are passed in `r1`–`r5` and the result is returned in `r0`. Because a helper's implementation is internal to the kernel, our formalization abstracts the entire call through a *call oracle* (Section 3), capturing only the observable contract: callee registers are preserved and the helper may not corrupt caller-owned memory.

Before execution, every program must pass the *kernel verifier*, which enforces memory safety, register initialization, and termination. The verifier rejects programs that contain back-edges (which could cause infinite loops), use uninitialized registers, or access memory outside sanctioned regions. However, the verifier's correctness is not formally proved: it is a complex heuristic analysis whose soundness bugs have been exploited in practice [14]. We leave the task of formalizing the eBPF verifier for future work.

2.2 Hoare And Separation Logic

Hoare logic [8] associates a triple $\{P\}\ C\ @\ \mathsf{pc}\ \{Q\}$ with a program C, program counter pc, precondition P, and postcondition Q, expressing partial correctness: if P holds initially and C terminates, then Q holds at termination.

Separation logic [15, 17] extends Hoare logic with the *separating conjunction* $P \star Q$, asserting that the heap splits into two disjoint parts satisfying P and Q respectively. The *points-to* assertion $p \hookrightarrow a$ states that address p holds value a and that the current heap owns exactly that cell.

As an example, consider the program $swap(p, q)$ that exchanges the values stored at two heap addresses. Writing $[e]$ for a load from address e and $[e] := v$ for a store of value v, the program consists of three assignments:

$$swap(p, q) \triangleq t := [p]; \quad [p] := [q]; \quad [q] := t$$

The first assignment saves the current value at p into a temporary t. The second overwrites p with the current value at q. The third writes the saved value into q, completing the exchange.

We prove $\{p \hookrightarrow a \star q \hookrightarrow b\}\ swap(p, q)\ @\ \mathsf{pc}\ \{p \hookrightarrow b \star q \hookrightarrow a\}$ by annotating each program point with the assertion holding there and identifying the rule applied at each step:

$$\begin{array}{ll}
\{p \hookrightarrow a \star q \hookrightarrow b\} & \\
\quad t := [p]; & \textsc{Load} \\
\{p \hookrightarrow a \star q \hookrightarrow b \star t = a\} & \\
\quad [p] := [q]; & \textsc{Load}; \textsc{Store} \\
\{p \hookrightarrow b \star q \hookrightarrow b \star t = a\} & \\
\quad [q] := t & \textsc{Store} \\
\{p \hookrightarrow b \star q \hookrightarrow a \star t = a\} & \\
 & \textsc{Consequence} \\
\{p \hookrightarrow b \star q \hookrightarrow a\} &
\end{array}$$

First, Load at $t := [p]$ certifies ownership via $p \hookrightarrow a$ and records $t = a$ as a pure fact; the heap survives unchanged. At $[p] := [q]$, a Load+Store pair first reads b from q (ownership from $q \hookrightarrow b$) and then writes b to p (updating $p \hookrightarrow a$ to $p \hookrightarrow b$); $q \hookrightarrow b$ is unaffected.

Next, Store at $[q] := t$ writes the saved value $t = a$ into q (ownership from $q \hookrightarrow b$), yielding $q \hookrightarrow a$. The cell $p \hookrightarrow b$ is untouched; $t = a$ remains as a pure fact.

Finally, Consequence drops the pure fact $t = a$, which was needed only to track the intermediate value and plays no role in the postcondition, yielding $p \hookrightarrow b \star q \hookrightarrow a$. The separating conjunction also ensures $p \neq q$ implicitly throughout: aliasing is structurally excluded, so no explicit hypothesis is needed.

The central *frame rule*

$$\frac{\{P\}\ C\ @\ \mathsf{pc}\ \{Q\}}{\{P \star R\}\ C\ @\ \mathsf{pc}\ \{Q \star R\}}\ \textsc{Frame}$$

enables local reasoning: a proof about heap region P remains valid in the presence of any disjoint frame R. Applied to the swap example, if a third address r (disjoint from p and q) holds value c, the frame rule immediately yields

$$\{\ p \hookrightarrow a \star q \hookrightarrow b \star r \hookrightarrow c\ \}\ swap(p, q) @ \mathsf{pc}\ \{\ p \hookrightarrow b \star q \hookrightarrow a \star r \hookrightarrow c\ \}$$

without re-examining *swap*: the frame $r \hookrightarrow c$ passes through untouched because *swap* only modifies p and q. In the eBPF setting the frame rule requires an additional condition on R specific to the eBPF register model, explained in Section 4.

3 Machine Model And Operational Semantics

3.1 Machine State

In this section, we define the eBPF machine model and its operational semantics. The machine state is a Lean 4 record:

$$\sigma = \{\ \mathtt{regs} : \mathtt{RegFile},\ \ \mathtt{mem} : \mathtt{Memory},\ \ \mathtt{pc} : \mathbb{N}\ \}$$

where `RegFile = Reg → BitVec 64` maps each register to a 64-bit word, and `Memory = BitVec 64 → BitVec 8` is a *total* byte-addressed memory. The register type `Reg` has constructors `r0`, …, `r10` with decidable equality. We write $\sigma.\mathtt{regs}$, $\sigma.\mathtt{mem}$, and $\sigma.\mathsf{pc}$ for the three components, and $rf\,[\mathtt{r} \mapsto v]$ (notation: $rf.\mathtt{set\ r}\ v$) for register update.

Using a total memory deviates from the actual eBPF execution model, where the kernel verifier's pointer-type analysis prevents out-of-bounds accesses at load time. In our formalization, we do not enforce memory safety at the operational level; instead, the

separation-logic precondition plays that role: a load or store can only be verified when the precondition asserts ownership of the relevant bytes, so a program that admits an `HTriple` derivation is safe with respect to the *owned* heap region, independently of the kernel verifier's analysis.

3.2 Instruction Set

A *program* is an array of instructions (`Program = Array Instr`). The inductive type `Instr` has one constructor per encoding class. The *ALU* class comprises `alu64` and `alu32` for binary operations on 64- and 32-bit registers, `neg64`/`neg32` for negation, and `endian` for byte-swapping at 16, 32, or 64 bits; the source operand is either a register or a sign-extended 32-bit immediate. The *memory* class provides `load` and `store` instructions parameterized by a transfer size $sz \in \{\texttt{byte}, \texttt{half}, \texttt{word}, \texttt{dword}\}$, which determines how many bytes (1, 2, 4, or 8, respectively) are read or written at the effective address $\texttt{regs}[r] + \texttt{signExt}(off)$; the class also includes `lddw`, which loads a 64-bit immediate and advances the PC by 2. The *control-flow* class includes `ja` for unconditional jumps, `jmp64` and `jmp32` for conditional branches supporting 11 comparison operators in 64- and 32-bit variants, and `exit` which halts execution with the return value in `r0`. The *helper-call* class has a single constructor `call` fid, which invokes the kernel helper identified by fid; arguments are passed in `r1`–`r5` and registers `r6`–`r10` are callee-saved. Finally, the *atomic* class provides a non-interruptible read-modify-write over the operations `add`, `or`, `and`, `xor`, `xchg`, and `cmpxchg`, with optional fetch-and-modify semantics.

3.3 Small-Step Semantics

The small-step relation `Step` *prog* σ σ' is an inductive Lean 4 proposition with one constructor per instruction class. Selected rules are shown below, where `evalAlu64` *op* v_1 v_2 and `evalJmp64` *op* v_1 v_2 are pure functions on `BitVec 64`, and `evalSrc` *rf s* extracts the source value ($rf[r]$ or $\texttt{signExt}(k)$).

We start with instruction Exit:

$$\frac{prog[\texttt{pc}]? = \texttt{some exit}}{\texttt{Step}\ prog\ \sigma\ \sigma} \textsc{Exit}$$

which halts the virtual machine in place; the return value is in `r0`. Next, we consider the semantics for a general ALU instruction:

$$\frac{\begin{array}{c} prog[\texttt{pc}]? = \texttt{some alu64}\ op\ dst\ src \\ v \leftarrow \texttt{evalAlu64}\ op\ (\texttt{regs}[dst])\ (\texttt{evalSrc}\ src) \\ \sigma' = \sigma[\texttt{pc} \mathrel{+}= 1, regs[dst] \leftarrow v] \end{array}}{\texttt{Step}\ prog\ \sigma\ \sigma'} \textsc{Alu64}$$

The result v is written into dst; memory is unchanged. The 32-bit variant uses `evalAlu32` and zero-extends the result.

We now consider the Load rule, which specifies how the machine reads sz bytes from memory at the effective address computed by adding the sign-extended offset to the base register and storing the result in dst:

$$\frac{\begin{array}{c} prog[\texttt{pc}]? = \texttt{some load}\ sz\ dst\ src\_reg\ off \\ addr = \texttt{regs}[src\_reg] + \texttt{signExt}(off) \\ \texttt{regs}[dst] \leftarrow \texttt{readMem mem}\ addr\ sz \\ \sigma' = \sigma[\texttt{pc} \mathrel{+}= 1,\ \texttt{regs}[dst]] \end{array}}{\texttt{Step}\ prog\ \sigma\ \sigma'} \textsc{Load}$$

`readMem` reads sz consecutive bytes in little-endian order and zero-extends the result to 64 bits; memory is left unchanged. Since `Memory` is a total function, every address is always readable at the semantic level; ownership and bounds are enforced by the separation-logic precondition rather than by the operational rule. The Store rule is symmetric: it uses `writeMem` to update sz bytes at the effective address and leaves all registers unchanged.

Conditional branches require two rules because the transition depends on whether the comparison evaluates to true or false. Jmp64-T fires when `evalJmp64` returns `true` and redirects control to $\texttt{pc} + 1 + off$:

$$\frac{\begin{array}{c} prog[\texttt{pc}]? = \texttt{some jmp64}\ op\ dst\ src\ off \\ \texttt{evalJmp64}\ op\ \texttt{regs}[dst]\ (\texttt{evalSrc}\ src) = \texttt{true} \end{array}}{\texttt{Step}\ prog\ \sigma\ \sigma[\texttt{pc} := \texttt{pc} + 1 + off]} \textsc{Jmp64-T}$$

When the comparison yields `false`, Jmp64-F falls through to the next instruction:

$$\frac{\begin{array}{c} prog[\texttt{pc}]? = \texttt{some jmp64}\ op\ dst\ src\ off \\ \texttt{evalJmp64}\ op\ \texttt{regs}[dst]\ (\texttt{evalSrc}\ src) = \texttt{false} \end{array}}{\texttt{Step}\ prog\ \sigma\ \sigma[\texttt{pc} \mathrel{+}= 1]} \textsc{Jmp64-F}$$

The two rules are mutually exclusive by the determinism of `evalJmp64`. Neither rule modifies registers or memory; only the program counter changes. The Jmp32 variant applies `evalJmp32` to the lower 32 bits of the operands, and Ja is the degenerate case that always takes the jump.

We model helper calls through a *call oracle*

$$\begin{array}{c} \texttt{CallOracle} : \texttt{BitVec 32} \to \texttt{RegFile} \to \\ \texttt{Memory} \to (\texttt{RegFile} \times \texttt{Memory}) \end{array}$$

since a helper's implementation is internal to the kernel and invisible to the formalization. The oracle abstracts the effect of each helper as an arbitrary function of the function identifier, the register file, and the memory. We require the oracle to satisfy `OracleOk`, the contract that callee-saved registers `r6`–`r10` are preserved across the call. The Call rule applies the oracle to produce the post-call register file and memory, and advances the PC by 1. A footprint fp records which memory addresses were modified by the call; it is consumed by `HeapSafe` (Section 5) to certify that the oracle does not corrupt the caller's owned heap region.

3.4 Multi-Step Runs

We define multi-step execution as the reflexive-transitive closure of `Step`, given by the inductive family `Steps` *prog* : `State` → `State` → `Type`:

$$\frac{}{\texttt{Steps}\ prog\ \sigma\ \sigma} \textsc{Refl} \qquad \frac{\texttt{Step}\ prog\ \sigma\ \sigma'' \quad \texttt{Steps}\ prog\ \sigma''\ \sigma'}{\texttt{Steps}\ prog\ \sigma\ \sigma'} \textsc{Step}$$

The Refl constructor witnesses the trivial zero-step execution in which the initial and final states coincide. The Step constructor prepends one transition: given a single step from σ to an intermediate state σ'' and a run from σ'' to σ', it produces a run from σ to σ'. Together they give the standard reflexive-transitive closure, but as an inductive family in `Type` rather than `Prop`. Note that this distinction is essential: a `Prop` derivation can only be eliminated into another `Prop`, so its structure cannot be inspected; living in `Type` allows the `HeapSafe` predicate (Section 5) to recurse over the

derivation tree and examine each Call step to verify that helper calls do not corrupt the caller's owned memory.

We say that a program terminates from an initial state s_0 when there exists a reachable state whose program counter points to an `exit` instruction:

$$\texttt{Terminates}\ \mathit{prog}\ s_0 \triangleq \exists s.\ \texttt{Nonempty}\ (\texttt{Steps}\ \mathit{prog}\ s_0\ s) \wedge \mathit{prog}[s.\texttt{pc}]? = \texttt{some exit}$$

The use of `Nonempty` collapses the `Type`-valued `Steps` evidence into a `Prop`, so that `Terminates` can appear in theorem statements without carrying a concrete derivation tree. The observable result of a terminating run is the value of register `r0` in the final state s.

4 Separation Logic And Hoare Triples

4.1 Partial Heaps

We now introduce the separation-logic infrastructure. The key notion is that of a *partial heap* $\texttt{Heap} = \texttt{BitVec 64} \to \texttt{Option (BitVec 8)}$, where `some` b means the heap owns that byte with value b and `none` means unowned. Four operations are central, where h, h_1, h_2 denote arbitrary partial heaps, m denotes the flat physical memory, and $m(a)$ denotes the byte stored at address a:

$$\begin{array}{lcl} h_1 \perp h_2 & \triangleq & \forall a.\ h_1(a) = \texttt{none} \ \vee\ h_2(a) = \texttt{none} \\ (h_1 \uplus h_2)(a) & \triangleq & h_1(a)\ \texttt{orElse}\ h_2(a) \\ h\ \texttt{Consistent}\ m & \triangleq & \forall a\ b.\ h(a) = \texttt{some}\ b \Rightarrow m(a) = b \\ h\ \texttt{empty} & \triangleq & \forall a.\ h(a) = \texttt{none} \end{array}$$

Disjointness, written $h_1 \perp h_2$, ensures that two heaps do not share ownership of any address, making separating conjunction $P \star Q$ meaningful. Union, written $h_1 \uplus h_2$, combines two disjoint heaps pointwise; it bridges the precondition heap h_1 and frame heap h_2 into the total owned region. Consistency relates the abstract heap to the flat physical memory m: every owned byte must agree with the byte at the same address in m, bridging the separation-logic and operational-semantics layers. The empty heap is the identity of $\uplus$ and $\mathbf{emp}\ rf\ h \triangleq h\ \texttt{empty}$.

The operation `readHeap` $h\ addr\ sz$ is all-or-nothing: it returns `some` v only when the heap owns all sz bytes of the chunk, ensuring a load can only be verified when the precondition asserts full ownership. Operation `writeHeap` replaces byte values without changing the ownership domain, so stores preserve the ownership footprint.

4.2 Assertions

An *assertion* is a predicate over a register file and a partial heap:

$$\mathbf{Assert} \triangleq \texttt{RegFile} \to \texttt{Heap} \to \texttt{Prop}$$

The program counter is tracked by the triple index `pc` and excluded from assertions; entailment $P \vdash_a Q$ is pointwise implication.

`emp` holds when the heap is entirely empty and serves as the identity of $\star$. `heapPts` $addr\ sz\ v$ holds when the heap owns *exactly* the sz-byte chunk at $addr$ and reading it little-endian yields v; precise ownership prevents hidden aliasing. `regPts` $r\ v$ constrains register r without affecting the heap; `Assert.pure` P and `Assert.regs` P lift propositions over registers.

$$(P \star Q)\ rf\ h \triangleq \exists h_1\ h_2.\ h_1 \perp h_2 \wedge h_1 \uplus h_2 = h \wedge P\ rf\ h_1 \wedge Q\ rf\ h_2$$

Both conjuncts receive the same register file rf because registers are not spatially split; frames that mention registers would be invalidated by ALU steps, so only heap-only frames are permitted. The magic wand $(P \mathrel{-\!\!\star} Q)\ rf\ h \triangleq \forall h'.\ h \perp h' \Rightarrow P\ rf\ h' \Rightarrow Q\ rf\ (h \uplus h')$ completes the separation-logic vocabulary.

The Frame rule requires `Assert.HeapOnly` R:

$$\texttt{Assert.HeapOnly}\ R \triangleq \forall\ rf\ rf'\ h.\ R\ rf\ h \Rightarrow R\ rf'\ h$$

Since R does not depend on registers, ALU instructions cannot falsify it. All spatial assertions (`emp`, `heapPts`, and their conjunctions) are heap-only.

4.3 Weakest-Precondition Transformers

The *weakest precondition* (WP) of an instruction with respect to a postcondition Q is the least restrictive precondition that guarantees Q after executing that instruction. WP transformers are the computational engine behind backward verification: instead of forward symbolic execution (which must enumerate program states), WP propagation starts at the desired postcondition and computes, one instruction at a time, the exact conditions that the initial state must satisfy for the postcondition to hold. In our setting each transformer is a higher-order function of type **Assert** → **Assert** (or **Assert** → **Assert** → **Assert** for branching instructions), making it a first-class Lean 4 term that the type-checker can reduce and the proof tactics can rewrite. We present the key transformers below; the remaining instruction classes follow the same pattern.

We start with the WP for ALU instructions:

$$\mathrm{wp}^{op,dst,src}_{\texttt{alu64}}\ Q = \lambda\ rf\ h.\ \begin{cases} Q\ (rf[dst \mapsto v])\ h\ \textbf{where} \\ \quad v_0 = \texttt{evalAlu64}\ op\ (rf[dst])\ v_1 \\ \quad v_1 = (\texttt{evalSrc}\ rf\ src) \end{cases}$$

The ALU transformer is a straightforward substitution: it replaces dst in the postcondition Q with the computed value v, passing the heap through unchanged. No ownership condition is imposed because ALU instructions never touch memory. The 32-bit variant $\mathrm{wp}_{\texttt{alu32}}$ is identical except that the result is zero-extended to fill the upper 32 bits of dst.

Next, memory load:

$$\mathrm{wp}^{sz,dst,r,off}_{\texttt{load}}\ Q\ =\ \lambda\ rf\ h.\ \exists v.\ \begin{cases} \texttt{readHeap}\ h\ (rf[r] + v_0)\ sz = s_1 \wedge \\ Q\ (rf[dst \mapsto v])\ h\ \textbf{where} \\ \quad v_0 = \texttt{signExt}\ off \\ \quad s_1 = \texttt{some}\ v \end{cases}$$

The load transformer introduces an existential witness v: the precondition requires the heap to own the full sz-byte chunk at the effective address and that reading it yields v, after which the postcondition holds with dst bound to v. Ownership is enforced via `readHeap`, which returns `none` for any unowned byte, making ownership a necessary condition for verification. This directly encodes the requirement that eBPF programs may only read from memory they own.

The WP transformer for store:

$$\begin{array}{rcl} \mathrm{wp}^{sz,dst,off,s}_{\texttt{store}}\ Q & = & \lambda\ rf\ h.\ \textbf{let}\ addr = rf[dst] + \texttt{signExt}\ off; \\ & & \qquad val = \texttt{evalSrc}\ rf\ s \\ & & \textbf{in}\ \texttt{readHeap}\ h\ addr\ sz \neq \texttt{none} \wedge \\ & & \quad Q\ rf\ (\texttt{writeHeap}\ h\ addr\ sz\ val) \end{array}$$

The store transformer requires the target chunk to already be owned (`readHeap` $\neq$ `none`) and threads the updated heap through `writeHeap` into the postcondition. Note that `writeHeap` only updates values at already-owned addresses, so stores are confined to allocated regions and the frame invariant is maintained across writes.

The transformer for jumps is as follows:

$$\mathrm{wp}^{op,dst,src}_{\texttt{jmp64}}\ Q_T\ Q_F = \lambda\ rf\ h.\begin{cases}(c_0 \Rightarrow Q_T\ rf\ h)\ \wedge\ (c_1 \Rightarrow Q_F\ rf\ h)\\ \textbf{where}\\ \quad v_0 = \texttt{evalSrc}\ rf\ src\\ \quad r = rf[dst]\\ \quad c_0 = (\texttt{evalJmp64}\ op\ r\ v_0 = \texttt{true})\\ \quad c_1 = (\texttt{evalJmp64}\ op\ r\ v_0 = \texttt{false})\end{cases}$$

Branching instructions take *two* postconditions: Q_T for the taken branch and Q_F for the fall-through. The transformer produces a conjunction of two implications, one per outcome, so the precondition must be strong enough to guarantee both Q_T when the condition is true and Q_F when it is false. Neither the register file nor the heap is modified by the branch itself; only the program counter changes.

The transformer for calls is as follows:

$$\mathrm{wp}_{\texttt{call}}\ Q\ =\ \lambda\ rf\ h.\ \forall rf'\ m'.\ h\ \texttt{Consistent}\ m'\ \Rightarrow\\ (\forall r \in \{\texttt{r6}\ldots\texttt{r10}\}.\ rf'[r] = rf[r])\ \Rightarrow\ Q\ rf'\ h$$

The call transformer is the most conservative: it universally quantifies over all post-call register files rf' and memories m', constraining only that callee-saved registers `r6–r10` are preserved and that m' remains consistent with the owned heap h. The postcondition must then hold for any such outcome. Note that `OracleOk` alone does not imply heap non-corruption; the `HeapSafe` predicate (Section 5) certifies this per step.

4.4 Separation Logic Rules

We define the inductive proposition `HTriple` *prog* `pc` P Q (written $\{P\}$ *prog* @ `pc` $\{Q\}$) to represent partial correctness: any terminated execution from a precondition-satisfying state reaches the postcondition. The rules are presented below, each accompanied by an explanation.

$$\frac{prog[\texttt{pc}]? = \texttt{some exit}}{\{Q\}\ prog\ @\ \texttt{pc}\ \{Q\}}\ \textsc{Exit}$$

The Exit rule is the unique base case of every `HTriple` derivation: every proof of a triple must eventually reach this rule. It fires only when the instruction at `pc` is `exit`, meaning execution has reached its terminal point. Because no further steps are taken, whatever assertion Q holds at that moment is simultaneously the precondition and the postcondition. The choice of Q is free: it is supplied by the context, and the return value observable to the caller is in register `r0`. In the backward proof methodology, this rule is applied first, at the final program counter, and the derivation proceeds toward lower PCs.

$$\frac{P' \vdash_a P \quad \{P\}\ prog\ @\ \texttt{pc}\ \{Q\} \quad Q \vdash_a Q'}{\{P'\}\ prog\ @\ \texttt{pc}\ \{Q'\}}\ \textsc{Consequence}$$

The Consequence rule is the standard structural rule that adapts a proved triple to a different specification without inspecting the program. Given $\{P\}$ *prog* @ `pc` $\{Q\}$, one obtains $\{P'\}$ *prog* @ `pc` $\{Q'\}$ whenever $P' \vdash_a P$ (the new precondition is no weaker than the original) and $Q \vdash_a Q'$ (the new postcondition is no stronger). Entailment $P \vdash_a Q$ is defined pointwise: $\forall rf\ h.\ P\ rf\ h \Rightarrow Q\ rf\ h$. In practice, each WP axiom produces a syntactically precise precondition, such as $\exists v.\ \texttt{readHeap}\ h\ addr\ sz = \texttt{some}\ v \wedge R\ (rf[dst \mapsto v])\ h$; Consequence then converts it to the user's intended assertion, e.g., a named predicate. This rule is also used to strengthen a precondition when the user knows additional facts that the WP transformer does not exploit.

$$\frac{\texttt{HeapOnly}\ R \quad \{P\}\ prog\ @\ \texttt{pc}\ \{Q\}}{\{P \star R\}\ prog\ @\ \texttt{pc}\ \{Q \star R\}}\ \textsc{Frame}$$

The Frame rule is the central modularity rule of separation logic and the primary source of reuse in the formalization. Starting from a local proof $\{P\}$ *prog* @ `pc` $\{Q\}$ that reasons only about a portion of memory, the rule lifts it to any larger state that additionally satisfies a disjoint frame R, yielding $\{P \star R\}$ *prog* @ `pc` $\{Q \star R\}$. The frame R is *exactly preserved* in the postcondition: it is not merely some assertion disjoint from Q, but the same R that appeared in the precondition, meaning the program does not modify any of R's cells. The hypothesis `HeapOnly` R is essential and is specific to the eBPF setting: because ALU instructions freely modify registers, a frame that mentions specific register values would be invalidated by intervening arithmetic steps. Restricting R to heap-only assertions (those that do not depend on the register file) ensures the frame survives every instruction in the program. All spatial assertions (`emp`, `heapPts`, and their separating conjunctions) are heap-only by construction.

$$\frac{\{P_1\}\ prog\ @\ \texttt{pc}\ \{Q\} \quad \{P_2\}\ prog\ @\ \texttt{pc}\ \{Q\}}{\{P_1 \vee P_2\}\ prog\ @\ \texttt{pc}\ \{Q\}}\ \textsc{Disj}$$

The Disj rule handles disjunctive preconditions: if the triple holds independently for each disjunct P_1 and P_2, it also holds when the initial state satisfies either one. In practice, conditional branches are typically handled directly by Jmp64, which generates separate continuation goals for each outcome; Disj is most useful when a user-introduced disjunction (from an explicit case split in a larger proof) must be discharged, or when the two branch postconditions need to be unified under a common triple. A companion Exists rule handles existential preconditions analogously.

$$\frac{prog[\texttt{pc}]? = \texttt{some alu64}\ op\ dst\ src \quad \{R\}\ prog\ @\ \texttt{pc} + 1\ \{Q\}}{\{\mathrm{wp}^{op,dst,src}_{\texttt{alu64}}\ R\}\ prog\ @\ \texttt{pc}\ \{Q\}}\ \textsc{Alu64}$$

The Alu64 rule is the backward axiom for 64-bit arithmetic and logic instructions. The single premise $\{R\}$ *prog* @ `pc` + 1 $\{Q\}$ requires a proof for the rest of the program starting at `pc` + 1; the conclusion's precondition is automatically generated by the WP transformer, which substitutes into R the value that *dst* will hold after executing `alu64` *op dst src*. Since ALU instructions never modify memory, the heap h passes through unchanged and no ownership obligation is introduced. The rule is *compositional*: once a triple for the suffix starting at `pc` + 1 is available, the rule for the current instruction extends it by one step toward the program entry. The 32-bit variant Alu32 follows the same pattern, additionally zero-extending the 32-bit result into the upper half of *dst*.

$$\frac{prog[\mathsf{pc}]? = \mathtt{some\ load}\ sz\ dst\ r\ off \quad \{R\}\ prog\ @\ \mathsf{pc}+1\ \{Q\}}{\{\mathrm{wp}_{\mathtt{load}}^{sz,dst,r,off}\ R\}\ prog\ @\ \mathsf{pc}\ \{Q\}}\ \textsc{Load}$$

The Load rule is the backward axiom for memory read instructions. The WP transformer $\mathrm{wp}_{\mathtt{load}}^{sz,dst,r,off}$ produces a precondition that (i) requires the heap to own the full sz-byte chunk at the effective address $rf[r]$ + `signExt`(off), and (ii) binds the loaded value v as an existential witness in the continuation precondition R (with dst set to v). The ownership check is enforced by `readHeap`, which returns `none` if any of the sz bytes is unowned: a load instruction cannot be verified unless the precondition asserts full ownership of the chunk. The heap is unchanged after the load; only the destination register dst is updated, so the same heap h is threaded into the continuation.

$$\frac{prog[\mathsf{pc}]? = \mathtt{some\ store}\ sz\ dst\ off\ s \quad \{R\}\ prog\ @\ \mathsf{pc}+1\ \{Q\}}{\{\mathrm{wp}_{\mathtt{store}}^{sz,dst,off,s}\ R\}\ prog\ @\ \mathsf{pc}\ \{Q\}}\ \textsc{Store}$$

The Store rule is the backward axiom for memory write instructions. $\mathrm{wp}_{\mathtt{store}}^{sz,dst,off,s}$ requires that the target chunk is already owned (`readHeap` $h\ addr\ sz \neq$ `none`) and passes the continuation R the heap obtained by updating the chunk's value via `writeHeap`. The register file is not changed by a store, so the same rf is threaded through. Requiring prior ownership is what prevents writes to arbitrary unowned addresses; combined with the fact that `writeHeap` only changes byte *values* at already-owned addresses (never adding or removing ownership), the frame invariant is preserved: a store within the owned footprint cannot disturb any disjoint frame heap, keeping Frame sound in the presence of writes.

$$\frac{\begin{array}{c}prog[\mathsf{pc}]? = \mathtt{some\ jmp64}\ op\ dst\ src\ off \\ \mathsf{pc}' = \mathsf{pc}+1+off \\ \{R_T\}\ prog\ @\ \mathsf{pc}'\ \{Q\} \quad \{R_F\}\ prog\ @\ \mathsf{pc}+1\ \{Q\}\end{array}}{\{\mathrm{wp}_{\mathtt{jmp64}}^{op,dst,src}\ R_T\ R_F\}\ prog\ @\ \mathsf{pc}\ \{Q\}}\ \textsc{Jmp64}$$

The Jmp64 rule handles conditional branches. Because a branch can go to either of two program counters, two continuation triples are required: $\{R_T\}\ prog\ @\ \mathsf{pc}'\ \{Q\}$ for the taken branch (target $\mathsf{pc}' = \mathsf{pc}+1+off$) and $\{R_F\}\ prog\ @\ \mathsf{pc}+1\ \{Q\}$ for the fall-through. The WP transformer $\mathrm{wp}_{\mathtt{jmp64}}^{op,dst,src}$ produces the precondition as a conjunction: when the branch condition evaluates to `true` the state must satisfy R_T, and when it evaluates to `false` it must satisfy R_F. Neither the register file nor the heap is modified by the branch; only the program counter changes. The `NoBackEdges` condition on the program guarantees $\mathsf{pc}' > \mathsf{pc}$, so both branches strictly advance the counter and the inductive proof tree is well-founded. The 32-bit variant Jmp32 is structurally identical, applying `evalJmp32` to the lower 32 bits of the operands.

$$\frac{prog[\mathsf{pc}]? = \mathtt{some\ call}\ fid \quad \{R\}\ prog\ @\ \mathsf{pc}+1\ \{Q\}}{\{\mathrm{wp}_{\mathtt{call}}\ R\}\ prog\ @\ \mathsf{pc}\ \{Q\}}\ \textsc{Call}$$

The Call rule is the most conservative axiom in the system, reflecting the fact that a helper's body is opaque: the formalization cannot inspect its implementation. The WP transformer $\mathrm{wp}_{\mathtt{call}}$ therefore universally quantifies over *all* post-call register files rf' and memories m' that satisfy two contracts: (i) the *callee-save invariant*, requiring $rf'[r] = rf[r]$ for every $r \in \{\mathtt{r6}, \ldots, \mathtt{r10}\}$; and (ii) *heap consistency*, requiring the new memory m' to agree with the owned heap h on every owned address, meaning the helper has not modified caller-owned cells. Any post-call state satisfying both contracts must establish the continuation precondition R. This over-approximation is sound: if the triple holds for every admissible outcome, it holds in particular for the actual outcome of any concrete oracle satisfying `OracleOk`. The ownership of h is thus an *invariant* of the call: the helper may allocate or modify memory outside h, but cannot touch the caller's owned region, which is exactly what the Frame rule and the `HeapSafe` predicate enforce at the semantic level. We omit the remaining rules (Lddw, Ja, Jmp32, Alu32, Neg64, Neg32, Endian) as they follow the same backward WP pattern. The Atomic rule is the notable exception: an atomic read-modify-write combines a load and a store in a single non-interruptible step, so its WP must simultaneously require ownership of the target chunk (as in $\mathrm{wp}_{\mathtt{load}}$) and thread the updated heap through to the continuation (as in $\mathrm{wp}_{\mathtt{store}}$); the complete definition is available in the repository [2].

5 Semantic Soundness

5.1 Call-Safety Predicate

We define the `HeapSafe` predicate to account for the fact that helper calls can modify memory. Formally, `HeapSafe` *prog h hsteps* is defined inductively over a `Steps` derivation: every call step in the sequence only modifies addresses outside the owned heap h:

$$\frac{}{\mathtt{HeapSafe}\ prog\ h\ (\mathtt{refl}\ \sigma)}\ \textsc{Refl}$$

$$\frac{\mathtt{HeapSafe}\ prog\ h\ hss \quad (\mathtt{isCall}\ hst \Rightarrow \forall a.\ \sigma'.\mathtt{mem}(a) \neq \sigma.\mathtt{mem}(a) \Rightarrow h(a) = \mathtt{none})}{\mathtt{HeapSafe}\ prog\ h\ (\mathtt{step}\ hst\ hss)}\ \textsc{Step}$$

Non-call steps (ALU, memory) are exempt from this condition: stores are permitted to overwrite h-owned cells because the store WP explicitly transfers ownership.

5.2 Frame-Aware Semantic Triple

While `HTriple` is an inductive syntactic judgment, we define the *semantic triple* `SemTriple` *prog* `pc` *P Q* to give it meaning directly in terms of the operational semantics. Informally, `SemTriple` says: for any terminating, call-safe execution starting in a precondition-satisfying state, the postcondition holds in the presence of an arbitrary disjoint frame heap. More precisely, the definition carries a silent frame heap h_2 as a parameter throughout the entire execution, making the frame rule an intrinsic property rather than a derived one.

$$\texttt{SemTriple}\ \mathit{prog}\ \texttt{pc}\ P\ Q \triangleq \forall \sigma\ h_1\ h_2.\ \sigma.\texttt{pc} = \texttt{pc} \Rightarrow P\ \sigma.\texttt{regs}\ h_1 \Rightarrow (h_1 \uplus h_2)\ \texttt{Consistent}\ \sigma.\texttt{mem} \Rightarrow h_1 \perp h_2 \Rightarrow \forall \sigma_f\ \mathit{hss} : \texttt{Steps}\ \mathit{prog}\ \sigma\ \sigma_f.\ \mathit{prog}[\sigma_f.\texttt{pc}]? = \texttt{some exit} \Rightarrow \texttt{HeapSafe}\ \mathit{prog}\ (h_1 \uplus h_2)\ \mathit{hss} \Rightarrow \exists h_f.\ Q\ \sigma_f.\texttt{regs}\ h_f \wedge (h_f \uplus h_2)\ \texttt{Consistent}\ \sigma_f.\texttt{mem} \wedge h_f \perp h_2$$

The definition opens with universal quantification over an initial state σ, a *local* heap h_1 on which P is evaluated, and an arbitrary *frame* heap h_2 never mentioned by P or Q. The heap h_1 is the portion of memory owned by the program being verified; h_2 is contributed by the surrounding context and represents memory the current triple does not touch. Four hypotheses set up the initial configuration: σ.pc = pc ensures execution starts at the right program point; $P\ \sigma$.regs h_1 asserts the precondition on the local heap; $(h_1 \uplus h_2)$ Consistent σ.mem bridges the abstract heap model with the flat physical memory; and $h_1 \perp h_2$ guarantees the two heaps do not overlap. We let notation $[P]$ *prog* @ pc $[Q]$ denote SemTriple P *prog* pc Q.

Rather than asserting the existence of a terminating run, the body universally quantifies over all Steps derivations *hss* from σ to any final state σ_f. Two additional conditions are imposed on each such run: σ_f must be positioned at an exit instruction, and the derivation must be call-safe (HeapSafe *prog* $(h_1 \uplus h_2)$ *hss*). This universal quantification makes SemTriple a statement of *partial correctness*: non-terminating programs and stuck programs satisfy it vacuously.

The conclusion is existential in a final local heap h_f: Q must hold on the final register file and h_f, and the frame conditions must be re-established: $h_f \perp h_2$ and $(h_f \uplus h_2)$ Consistent σ_f.mem. Crucially, h_2 is *exactly preserved*: the frame heap after execution is literally h_2, not merely some heap disjoint from h_f. This is precisely the semantic content of the frame rule: the surrounding context's memory is untouched by the verified program.

The syntactic judgment HTriple is what a user constructs when verifying a program; SemTriple is the universally quantified statement over the operational model that guarantees correctness. The soundness theorem htriple_sound (below) establishes that every HTriple derivation implies the corresponding SemTriple.

5.3 Main Soundness Theorem

The main soundness theorem is the central result of the formalization. It closes the gap between the syntactic derivation system HTriple and the operational model by guaranteeing that every triple that can be derived using the inference rules is also valid with respect to the small-step semantics. Without this theorem, the derivation rules would be a purely formal game with no connection to program behavior; with it, every verification performed using HTriple carries a machine-checked guarantee that the verified property holds in any actual execution of the program.

Theorem 5.1 (htriple_sound). *For any prog,* pc, *P, Q, we have that:*

$$\{P\}\ \mathit{prog}\ @\ \texttt{pc}\ \{Q\} \Rightarrow [P]\ \mathit{prog}\ @\ \texttt{pc}\ [Q]$$

Using inversion lemmas for each instruction class, the proof proceeds by induction on the $\{P\}$ *prog* @ pc $\{Q\}$ derivation. Each instruction case reduces to a lemma that matches the instruction's Step constructor and propagates the frame. The Frame case relies on the heap union associativity algebra. The Call case is the most involved: it uses the HeapSafe witness to confirm that the oracle leaves $h_1 \uplus h_2$-owned addresses intact, then applies the call WP's universal quantification over oracle outcomes.

6 Case Study: Verified IPv4/TCP Packet Filter

As a case study, we verify an IPv4/TCP packet-filter program against a separation-logic specification. The program accepts only IPv4 TCP packets, checking the Ethernet type (0x0800) at byte 12 and the IP protocol field (0x06) at byte 23. Register r1 holds the base address of the packet buffer (value 0), the heap owns the relevant Ethernet and IP header bytes, and the return value is in r0: 1 for accept, 0 for drop.

PC	Instruction	Comment
0	load half r2 r1 12	load Ethernet type
1	jmp64 jne r2 0x0800 +4	if not IPv4, jump to PC 6
2	load byte r2 r1 23	load IP protocol
3	jmp64 jne r2 0x06 +2	if not TCP, jump to PC 6
4	alu64 mov r0 1	accept
5	exit	
6	alu64 mov r0 0	drop
7	exit	

The specification is:

$$\mathit{tcpPre}\ \mathit{rf}\ h \triangleq \mathit{rf}[\texttt{r1}] = 0 \wedge \texttt{readHeap}\ h\ 12\ \texttt{half} = \texttt{some 0x0800} \wedge \texttt{readHeap}\ h\ 23\ \texttt{byte} = \texttt{some 0x06}$$

$$\mathit{tcpPost}\ \mathit{rf}\ \_ \triangleq \mathit{rf}[\texttt{r0}] = 1$$

The proof obligation is: given a packet buffer whose Ethernet type field (bytes 12–13) holds 0x0800 (IPv4) and whose IP protocol field (byte 23) holds 0x06 (TCP), the program always returns 1. The argument proceeds by case analysis on the two conditional checks.

We first examine the drop paths. Both jump instructions can redirect execution to PCs 6–7, which write 0 into r0 and exit. Since the postcondition requires r0 = 1, any execution that reaches the drop code would falsify it. However, *tcpPre* already asserts that the packet *is* IPv4 and *is* TCP, so neither jump condition can be true: the Ethernet-type check at PC 1 will always pass, and the protocol check at PC 3 will always pass. The drop sub-triple is therefore vacuously correct: its precondition is **False**, so it imposes no obligation.

Next, we consider the accept path. When both checks pass, PCs 4–5 write 1 into r0 and exit. The postcondition r0 = 1 holds unconditionally after this assignment, so the accept sub-triple holds for any incoming state.

We now trace the verification backward from PC 3. For execution to reach the accept path rather than the drop path, the IP protocol field must equal 0x06. The proof therefore requires that at PC 3 the value just loaded into r2 is exactly 0x06. Any other value would trigger the drop branch, which is vacuously safe but unreachable given *tcpPre*.

At PC 2, the machine reads a single byte from byte 23 of the packet buffer into r2. To guarantee the check at PC 3 will pass, the proof requires that the heap owns byte 23 of the buffer and that the

stored value is `0x06` (the IANA protocol number for TCP). This is exactly the second heap conjunct of *tcpPre*: `readHeap` h 23 `byte` = `some 0x06`.

Finally, we handle PCs 0–1. For execution to proceed to the protocol check rather than drop, the Ethernet type field must equal `0x0800` (IPv4). PC 0 reads the 16-bit Ethernet type from bytes 12–13 of the packet buffer into `r2`; the proof requires heap ownership of that region and that the value is `0x0800`. Together, the two load obligations demand that the packet buffer contains `0x0800` at bytes 12–13 *and* `0x06` at byte 23.

We conclude by noting that since `r1` = 0, the base address of the buffer is 0, so "byte 12" and "byte 23" are absolute heap addresses 12 and 23. The conjuncts of *tcpPre* directly assert ownership of those two addresses with the required values, discharging both obligations and completing the proof.

We now state the main result of this section:

Theorem 6.1 (`tcpFilter_correct`). *Let tcpFilter be the eBPF program for filtering IPv4 packets. Then, we have that:*

$$\{tcpPre\}\ tcpFilter\ @\ 0\ \{tcpPost\}$$

Corollary 6.2 (`tcpFilter_sem_sound`). *If tcpPre* σ.`regs` h_1, *the oracle is heap-safe, and the interpreter returns* σ_f, *then* σ_f[`r0`] = 1.

The repository [2] additionally contains a verified ARP packet filter (accepting frames with EtherType `0x0806`) that exercises the same Load/Jmp64/Exit rules in a shorter four-instruction program, demonstrating the reusability of the logic across different filter patterns.

7 Related Work

Formal eBPF semantics and verification. Nelson et al. [14] verify semantic preservation of the BPF JIT compiler in Linux using Serval, catching real JIT bugs; in contrast to our work, their scope is limited to JIT correctness, and no program-level separation logic is developed. Yuan et al. [25, 26] construct a formally verified eBPF virtual machine in Rocq for microcontrollers and IoT scenarios, proving memory safety under simplified runtime models; in contrast, we target a more realistic flat byte-addressed memory and provide a mechanically proved separation logic. The uBPF project [16] provides a reference userspace eBPF interpreter, and an active conformance test suite [10] documents divergences across JIT back-ends; these empirical efforts motivate but do not replace a machine-checked formal semantics. Ball et al. [4] apply thorough static analysis to device drivers, highlighting the same gap between informal kernel interfaces and verified behavior that motivates our work.

Separation logic: foundations and mechanization. Reynolds [15] introduced the frame rule and separating conjunction as the basis for local, compositional heap reasoning; Pym et al. [17] provide a comprehensive categorical account of the model theory. Tan and Appel [20] develop a compositional logic for control flow at the assembly level, showing that low-level code admits Hoare-style reasoning; our work adopts this perspective for the eBPF VM. Our key novelty is the `HeapOnly` restriction: because eBPF instructions freely modify registers, naïve separation over register-and-heap assertions is unsound; we isolate this by restricting frames to heap-only assertions and equipping the semantic triple with per-execution `HeapSafe` witnesses for helper calls.

Verified virtual machines and compilers. CompCert [9, 11] is the landmark verified C compiler in Rocq, establishing the template of a mechanized semantics as the foundation for compiler correctness proofs. KEVM [7] provides a complete EVM specification in K, enabling smart-contract verification; our work is the analogue for the eBPF kernel domain in Lean 4. Amin and Rompf [1] prove type soundness using definitional interpreters, demonstrating how an inductive semantics supports soundness theorems; our `htriple_sound` follows the same methodology for separation logic.

Mechanization in Lean 4. Lean 4 [13] serves both as a dependently typed functional programming language and as an interactive theorem prover with a unified elaboration mechanism. Its macro system [23] enables notation that keeps mechanized proofs readable. Our formalization exploits Lean 4's `native_decide` tactic to close concrete `BitVec` arithmetic goals (e.g., address offset computations in the packet filter proofs) by kernel evaluation, avoiding manual arithmetic lemmas. It is important to note that this comes at a cost: `native_decide` extends the trusted base to include the Lean compiler and runtime, so the proofs are admitted-free but rely on the correctness of kernel evaluation for `BitVec` goals.

8 Conclusion

We have presented a mechanized formalization of eBPF in Lean 4 comprising: a small-step inductive semantics over 64-bit registers and flat byte-addressed memory; a separation logic with a proved-sound frame rule using `HeapOnly` frames and `HeapSafe` call-safety witnesses; the soundness theorem `htriple_sound` (`HTriple` ⇒ `SemTriple`); and an end-to-end correctness proof for an IPv4/TCP packet-filter program.

As future work, we plan to: (a) formally present the certified interpreter results (fuel-based and fuel-free, with bridge theorems to `SemTriple`), deferred here for space; (b) extend the ISA to cover eBPF maps, tail calls, and XDP helpers; (c) automate the entailment proofs arising from Consequence via a separation-logic decision procedure; and (d) connect the formalization to the kernel verifier's type-and-pointer analysis to verify programs without requiring manual heap annotations.

Artifact Availability

The complete Lean 4 source of the formalization is available at the following repository:

https://github.com/lives-group/lean-ebpf

Use of Generative AI

Generative AI technologies (specifically Gemini) were used in the preparation of this work for grammatical review. The authors reviewed all AI-generated suggestions and take full responsibility for the content and validity of the final text.

References

[1] Nada Amin and Tiark Rompf. 2017. Type soundness proofs with definitional interpreters. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 666–679. doi:10.1145/3009837.3009866

[2] Anonymous. 2025. Executable Formal Specification of eBPF in Lean 4. OMITTED. Accessed on: July 20, 2025.

[3] Cilium Authors. 2024. BPF and XDP Reference Guide. https://docs.cilium.io/en/latest/bpf/. BPF and XDP Reference Guide - Cilium 1.16.0-dev documentation.

[4] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K Rajamani, and Abdullah Ustuner. 2006. Thorough static analysis of device drivers. *ACM SIGOPS Operating Systems Review* 40, 4 (2006), 73–85.

[5] Gilberto Bertin. 2017. XDP in practice: integrating XDP into our DDoS mitigation pipeline. In *Netdev 2.1 - Technical Conference on Linux Networking*. 1–5.

[6] Facebook. 2025. Katran source code repository. https://github.com/facebookincubator/katran.

[7] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Ștefănescu, and Grigore Roșu. 2018. KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. doi:10.1109/CSF.2018.00022

[8] Charles Antony Richard Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580.

[9] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. 2015. A Formally-Verified C Static Analyzer. In *POPL 2015: 42nd symposium Principles of Programming Languages*. ACM Press, 247–259. http://xavierleroy.org/publi/verasco-popl2015.pdf

[10] Alan Jowett. 2025. eBPF conformance tests. https://github.com/Alan-Jowett/bpf_conformance.

[11] Daniel Kästner, Ulrich Wünsche, Jörg Barrho, Marc Schlickling, Bernhard Schommer, Michael Schmidt, Christian Ferdinand, Xavier Leroy, and Sandrine Blazy. 2018. CompCert: Practical experience on integrating and qualifying a formally verified optimizing compiler. In *ERTS 2018: Embedded Real Time Software and Systems*. SEE. http://xavierleroy.org/publi/erts2018_compcert.pdf

[12] Steven McCanne and Van Jacobson. 1993. The BSD packet filter: a new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings* (San Diego, California) *(USENIX'93)*. USENIX Association, USA, 2.

[13] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15 , 2021, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 625–635. doi:10.1007/978-3-030-79876-5_37

[14] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. 2020. Specification and Verification in the Field: Applying Formal Methods to BPF Just-in-Time Compilers in the Linux Kernel. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 41–61.

[15] Peter O'Hearn, John Reynolds, and Hongseok Yang. 2001. Local reasoning about programs that alter data structures. In *International Workshop on Computer Science Logic*. Springer, 1–19.

[16] IOVisor Project. 2025. uBPF: A Userspace Virtual Machine for eBPF. https://github.com/iovisor/ubpf/tree/main/tests. Accessed on 2026-05-12.

[17] David Pym, Jonathan M Spring, and Peter O'Hearn. 2019. Why separation logic works. *Philosophy & Technology* 32, 3 (2019), 483–516.

[18] Liz Rice. 2023. *Learning eBPF: Programming the Linux Kernel for Enhanced Observability, Networking, and Security*. O'Reilly Media, Inc.

[19] Jay Schulist, Daniel Borkmann, and Alexei Starovoitov. 2025. Linux Socket Filtering aka Berkeley Packet Filter (BPF). https://www.kernel.org/doc/Documentation/networking/filter.txt. Accessed on 12/05/2026.

[20] Gang Tan and Andrew W Appel. 2006. A compositional logic for control flow. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 80–94.

[21] Dave Thaler. 2024. *BPF Instruction Set Architecture (ISA)*. RFC 9669. Internet Engineering Task Force. Standards Track.

[22] The Linux Kernel Community. 2024. Berkeley Packet Filter (BPF) Documentation. https://www.kernel.org/doc/html/latest/bpf/index.html. Accessed on 05/12/2026.

[23] Sebastian Ullrich and Leonardo de Moura. 2020. Beyond Notations: Hygienic Macro Expansion for Theorem Proving Languages. In *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-5, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12166)*. Springer International Publishing, 167–182. doi:10.1007/978-3-030-51054-1_10

[24] Marcos A. M. Vieira, Matheus S. Castanho, Racyus D. G. Pací fico, Elerson R. S. Santos, Eduardo P. M. Câmara Júnior, and Luiz F. M. Vieira. 2020. Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications. *ACM Comput. Surv.* 53, 1, Article 16 (Feb. 2020), 36 pages. doi:10.1145/3371038

[25] Shenghao Yuan, Frédéric Besson, and Jean-Pierre Talpin. 2024. End-to-end mechanized proof of a JIT-accelerated eBPF virtual machine for IoT. In *International Conference on Computer Aided Verification*. Springer, 325–347.

[26] Shenghao Yuan, Frédéric Besson, Jean-Pierre Talpin, Samuel Hym, Koen Zandberg, and Emmanuel Baccelli. 2022. End-to-end Mechanized Proof of an eBPF Virtual Machine for Micro-controllers. In *International Conference on Computer Aided Verification*. Springer, 293–316.