Hydra: Rich and Scalable Functional Verification of eBPF Deployments

Lucas Castanheira
lcas@cmu.edu
Carnegie Mellon University
USA

Theophilus A. Benson
theophilus@cmu.edu
Carnegie Mellon University
USA

Abstract

Production eBPF deployments now run as deep, dynamically composed *chains* of programs. Reasoning about the behavior of such chains is critical – e.g., an upstream program may silently overwrite a downstream's routing decision. Existing symbolic-execution verifiers collapse under path explosion when confronted with large programs or the combinatorial space of deployed chains; their helper models, manually developed and incomplete, further erode soundness.

We present Hydra, a verifier for eBPF deployments that (i) reframes verification at the granularity of *program graphs*, (ii) optimizes verification, with each program being efficiently transformed into path summaries that are reused across chains to answer graph-level queries; and (iii) creates accurate helper models, synthesized by LLMs and validated against the kernel via `BPF_PROG_TEST_RUN` + `kcov`. Across 12 programs and 27 synthetic program graphs, Hydra correctly verifies every chain with aggregate speedups over a monolithic baseline of at least ∼ 80× (up to 1,506×). Its helper models enumerate 2.5–6× more kernel-validated paths per helper than `ebpf-se`.

CCS Concepts

• **Software and its engineering → Software verification**.

Keywords

eBPF; symbolic execution; compositional verification

ACM Reference Format:
Lucas Castanheira and Theophilus A. Benson. 2026. Hydra: Rich and Scalable Functional Verification of eBPF Deployments. In *4th Workshop on eBPF and Kernel Extensions (eBPF'26), September 29-October 02, 2026, Prague, Czech Republic.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3837779.3838170



This work is licensed under a Creative Commons Attribution 4.0 International License.
eBPF'26, Prague, Czech Republic
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2911-9/2026/09
https://doi.org/10.1145/3837779.3838170

1 Introduction

Extended Berkeley Packet Filter (eBPF) lets operators safely customize kernel behavior at run-time, and large fleets now lean on it heavily: at Meta, half of all servers run at least 180 eBPF programs concurrently [12], and virtually every server runs at least ten; Netflix reports similarly pervasive fleet-wide eBPF deployments [19]. A single kernel event — a packet arriving at XDP, a socket transitioning state — dispatches not one program but a *chain* of them, with the ordering being manually coordinated across the fleet's infrastructure teams (e.g., networking, security, observability). Heterogeneity is widespread: programs are added to and removed from each chain for different goals; moreover, each program can typically have several deployed variants (e.g., older versions required for compatibility or even full ports to accommodate certain Linux versions). Ideally, a given program should operate correctly under any possible chain; in practice, the sheer diversity of deployed chains often surfaces unforeseen and problematic interactions (a problem well known in service-oriented architectures), which traditional unit testing is ill-equipped to catch (developers cannot possibly unit test against all possible chains) [2]: for example, a program might blindly and silently overwrite routing decisions made by predecessors. The in-kernel eBPF verifier offers no help here: it admits programs one at a time and checks only safety properties (memory access, bounded loops), not program semantics or cross-program interactions [22].

Automated software verification emerges as uniquely qualified to give operators proactive correctness guarantees about chains: systematically checking a property over every possible chain and input, and producing actionable counterexamples when a check fails. However, current eBPF verification efforts that are rich enough to verify such properties (e.g., `ebpf-se` [9] and DRACO [17], both based on symbolic execution (symbex)) have fundamental limitations with respect to scalability (neither tames well-known path explosion from large programs and large chains) and correctness (both rely on manually developed helper models to overcome symbex's inability to easily run kernel code such as BPF helpers).

In this paper, we introduce Hydra, a system to scalably and accurately verify the behavior of eBPF deployments. This poses three challenges: **(a)** *what* to verify, **(b)** speed, and **(c)**

accuracy. To handle **(a)**, Hydra starts with two abstraction shifts. First, it changes the verification scope from individual programs (as in prior symbex tools [9, 13, 17] and the in-kernel verifier itself [22]) to *graphs* of programs. Second, it identifies *inter-program observation assumptions* as the core issue that semantic verification of dynamic deployments needs to be geared towards: a developer writing an eBPF program assumes that the state observations it emits (e.g., "I write `skb->mark` so the kernel will route on it") reach the kernel intact and drive the intended semantic outcome. Once that program runs inside a chain, dataflow becomes shared, and the assumption can silently break — an intermediate program may overwrite `skb->mark` or drop the packet altogether (a failure mode Meta observed in production chains for the analogous `skops->reply` field [12]). Hydra recasts verification of dynamic eBPF deployments as *providing users (e.g., Dev/Ops) with guarantees that, for every deployed chain, the state observations relevant to their programs' correct behavior will hold.*

To address speed **(b)**, Hydra relies on two layers of optimization over standard symbolic execution: (i) a *graph-level* layer that applies *compositional symbolic execution* [1, 11]: a classic technique in which each unit of code is symbolically executed only once, in isolation, and whole-system analyses are composed from the resulting per-unit summaries. Here the units are eBPF programs, the summaries are per-path observations (e.g., writes to `skb->mark`), and composition is an SMT query that stitches individual program summaries according to the deployment graph's chains and verifies desired properties across them. Moreover, because single-program symbolic execution can still be cost-prohibitive for larger programs, Hydra introduces (ii) a *program-level*, eBPF-specific layer that applies query-aware path pruning during summary generation. For example, when the query is "will `skb->mark` be overwritten by any two chained programs?", exploration of paths that never edit `skb->mark` can be carefully pruned, resulting in faster summarization and smaller summaries for (i)'s SMT query to process.

To address coverage issues **(c)**, Hydra replaces manually developed BPF helper models commonly used in verification [9, 17] with helper models synthesized directly from kernel source by an LLM-driven loop and validated path-by-path against the actual kernel: each model-emitted path is replayed through `BPF_PROG_TEST_RUN` and checked against `kcov` coverage reports. The result is symbolic execution that runs with helper models whose path-level behavior is higher coverage and hallucination-free.

Hydra makes the following contributions:

- **Scalable Verification.** A two-layer optimization stack that accelerates symbolic execution at both the *graph* level (decomposing the verification problem into reusable per-program summaries stitched together via SMT) and the *program* level (pruning each program's CFG against the queried observation before symbex even runs) (§3.2–3.3).
- **Accurate eBPF Helper Models for Verification.** A kcov-grounded helper-modeling pipeline that synthesizes BPF helper models from verbatim kernel source by an LLM loop and validates every emitted path against the real kernel via `BPF_PROG_TEST_RUN` with `kcov`, closing the soundness gap left by manually developed BPF helper stubs (§3.4).
- **Prototype and Preliminary Results.** We implement Hydra and evaluate our example queries on several graphs of eBPF programs (drawn from open-source repositories including Mizar [6] and Meta's katran [18]).

We find Hydra's verdicts match the monolithic baseline on every chain, with aggregate wall-clock speedups between 80× and 1,506× depending on Hydra's deployment mode, and its helper models enumerate 2.5× to 6× more kernel-validated paths per helper than the manually developed models in `ebpf-se`.

2 Motivation

We give real-world queries that motivate Hydra's abstractions on verification. We then motivate the need for fast and accurate verification given (i) the growing complexity of real-world eBPF deployments and how it makes traditional symbolic execution infeasible, and (ii) the inaccuracy of existing helper models and how they compromise verification soundness.

Verification Needs. At a high level, inter-program verification aims to answer the question: *will the state observations that are semantically meaningful for a program's correct behavior hold under any possible chaining of the deployment?* We observed correctness-breaking scenarios in which seemingly harmless chaining creates issues that are hard both to predict and to diagnose; we illustrate them with the following example. Consider a chain of TC network programs: a firewall (`fw`), a load balancer (`lb`), and a measurement program (`stats`), owned by different teams and chained by operators to meet their deployment needs. No program is wrong in isolation; composed, many problematic interactions may occur depending on how the programs were implemented. For example: (i) if both `fw` and `lb` depend on writes to the same metadata (e.g., `skb->mark`) that they expect the kernel to later see and act upon, `lb` may silently overwrite `fw`'s writes and break `fw`'s intended semantics; (ii) if a subset of chains migrates to a new version of `lb` (e.g., to accommodate a new network header), unexpected divergences between deployed chains may arise: if `fw` is not updated to also accept the new header, it may silently drop those packets before `lbv2`

ever sees them. Worse, in the original chain, `lbv1` might have never dropped packets (common, since basic packet malformation would be spotted and dropped earlier by `fw`), but if `lbv2`'s new parsing code for the new header includes drops, then in some cases `fw` will accept a packet that `lbv2` drops (both a silent new behavior and a hazard if, for example, the firewall allocates state to track connections of accepted packets); (iii) operators might intend for `stats` to always run for `fw`-accepted packets, but the `lbv2` update can cause `fw`-accepted packets to be dropped before `stats` runs, breaking the operators' intent and silently biasing measurements. From these scenarios, we generalize an initial set of correctness queries/invariants that Hydra aims to verify:

No Overwrites. Certain programs depend on their written state persisting so that the kernel (or a downstream consumer) can act on it (scenario (i)). At Meta, for example, programs in a chain frequently clobber each other's updates to shared metadata [12] — overwriting `skops->reply` — causing the kernel to route or act on a stale value instead of the intended one. The currently-accepted remediation is to detect such events at runtime and report them to the teams responsible for the offending write, incurring large reproducibility and triage overheads. Ideally, verification would prove the property holds across every possible chain in the deployment, or surface the exact chain orderings and inputs on which it breaks. This query asks: *"can my write to a shared metadata field be overwritten by any program later in any possible chain of my deployment?"*

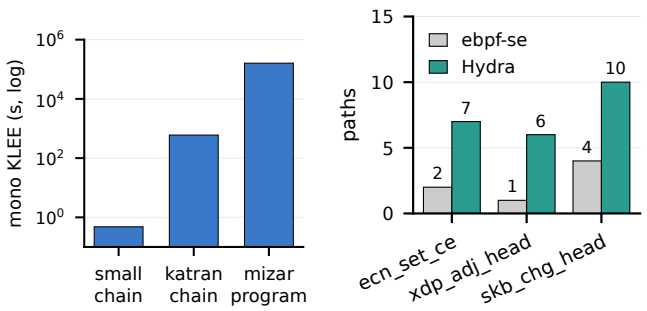
Parallel Invariance. A deployment often has many variants of a given program (e.g., A/B tests, staged fixes, and kernel-version ports, as in scenario (ii)). When adding a new version, it is hard to judge how it will change the chain's behavior: for strict fixes (changes meant to be behaviorally invisible, such as a security patch), it would be best if no downstream interfaces ever saw a difference. For A/B tests, it is important to understand if the differences are as expected or if the variant has unexpected side effects. For either, unit-tests are insufficient and might not flag any divergence. Ideally, verification would prove behavioral equivalence at every downstream observation point before deployment, or surface the exact inputs on which two variants diverge. This query asks: *"do all my co-deployed variants look identical to every downstream program that observes them? if not, how so?"*

Invariant Execution. eBPF programs may terminate the chain early (e.g., for fast drops and redirects). This creates the hazard that important downstream programs might be skipped (e.g., security auditing, DDoS detection, always-on observability). Sometimes this is semantically desirable and predictable (e.g., no need to waste load-balancing effort on a packet the firewall just dropped), and sometimes it is not (scenario (iii): an intermediate program drops a firewall-accepted packet before `stats` can run, silently biasing measurements). When chaining programs, instead of having to guess bad interactions or handwrite wrapper code to avoid them, operators can use verification to prove that a mandatory program always executes whenever a set of conditions is met (e.g., `fw` accepts ⇒ `stats` runs), or to surface a reproducible counterexample otherwise.

The Role of Symbolic Execution. The properties above are too rich (e.g., *for all input packets and all valid chain orderings, does the write to* `skb->mark` *reach the kernel intact?*, or *does there exist a chain ordering on which my compliance auditor is bypassed?*) to be captured by simple static analysis or unit tests: they quantify over all inputs and all permitted chain orderings, and a violation is only actionable together with the concrete input and ordering that triggers it. For such cases, symbolic execution is commonly used to reason about all possible paths through a program[1], and can be used to generate concrete BPF input (e.g., packet + underlying BPF map state) that triggers each specific path, together with the *path condition*: the constraint over inputs under which that path is taken. The total set of a program's paths lets us reason at the level of statements universally or existentially quantified over chain orderings and program inputs ($\forall$ chain. $\forall$ input. *property*), as well as produce concrete counterexamples when the property is violated (useful for fixing the violation). However, symbex is computationally expensive and cannot run kernel code such as helpers, because helpers execute inside the kernel against live kernel state while KLEE [5] interprets user-space LLVM IR; libraries like `libbpf-stubbed`, used by `ebpf-se` [9] and PIX [13], sidestep this by compiling BPF programs to LLVM IR and linking against a user-space implementation of `libbpf` whose helper functions are manually developed stubs that KLEE can symbolically execute. Hydra reuses `libbpf-stubbed` as the per-program symbex backend and replaces the stubs themselves with kcov-grounded helper models (§3.4).

Verification Complexity – Graph Depth and Branching: When modeled as a graph – each chain found in a deployment is a path from root to end node, each different program version is a branch in the path – eBPF deployments have significant depth and branching complexity: at Meta, half of all hosts run ≥ 180 programs, organized into chains, and a single program may span up to 10 concurrent variants accommodating up to 20 different kernel versions. Branching and depth are critical for verification complexity — even a modest depth-5 chain with 5 variants per slot enumerates 5^5=3,125 distinct chain materializations a complete verifier must consider. Existing verification tools [17] can be adapted to treat each materialization as an inlined monolithic program to be validated, however, this approach

[1]This path set is finite: the in-kernel verifier admits only programs whose loops are bounded.



(a) Monolithic KLEE wall-clock. **(b) Helper-path coverage.**

Figure 1: Speed and Accuracy Challenges: (a) monolithic symbex does not scale to either medium chains with real programs nor single large programs. (b) manually developed helper models (ebpf-se) leave several kernel paths uncovered compared to Hydra's kcov-validated helper models.

quickly becomes intractable due to path explosion (Fig. 1a): individual symbex on N chains composed of M katran-sized programs scales terribly — a single depth-5 katran chain is already costly (times out at 10 minutes); additionally, large codebases such as Mizar can make symbex problematic even for a single program (a single Mizar program takes days to verify using KLEE [5]). We must optimize both inter- and intra-program symbex to make verification tractable.

Verification Soundness – Kernel Code in Symbex: BPF programs can safely leverage rich functionality from the kernel through the use of helpers functions. For verification, these calls are crucial to understand the path each program takes (e.g., a helper failing might be the difference between an early packet drop vs a successful forwarding). Unfortunately, these calls are also deeply impractical to model. Most works [9, 13, 17, 24] use simplified models; `ebpf-se`, for example, declares manually written KLEE-friendly versions that broadly imitate the behavior of helpers for the common case but are limited in the coverage they can achieve (Fig. 1b): e.g., `ebpf-se`'s `bpf_xdp_adjust_head` omits error-handling codepaths – it has no bounds check and will always succeed, whereas the real kernel would return `-EINVAL` on multiple conditions. These discrepancies can magnify during symbex — e.g., a program that calls an insufficiently-modeled helper with gaps will itself have gaps, in the worst case, at each codepath that includes a call to the bad helper, thus compromising soundness.

3 Design

We now describe Hydra's workflow and the algorithms behind it.

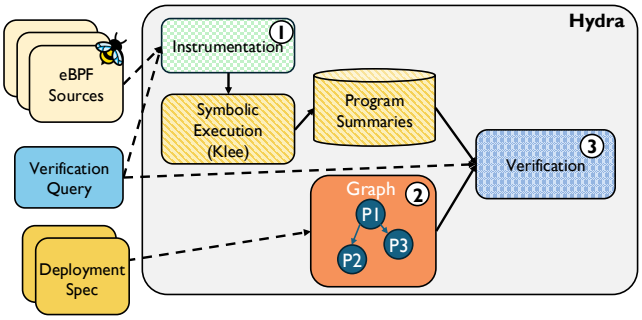


Figure 2: Hydra workflow

3.1 Hydra Workflow

Fig. 2 shows the workflow of Hydra, which takes as input a parameterized verification query (e.g., Table 1), a set of eBPF sources (with alternative versions of a program explicitly identified) and a deployment specification (i.e., the set of all manually-decided chains that can be active in production).

First, given the query of interest, Hydra instruments each individual program so that a symbolic-execution engine (SEE; KLEE in our prototype) can collect, for each of that program's paths, (i) the query-specific observations, (ii) the condition necessary to hit the path, and (iii) the path's returned verdict. The latter two are generally collected by any symbex framework, while (i) is specific to Hydra queries. Instrumented programs are then symbolically executed individually.

Second, Hydra assembles a unified deployment graph where nodes are programs, and the valid chainings (given by Dev/Ops) are paths through the graph — typically from a kernel hook entry to a chain leaf, with potentially multiple roots and sinks across the deployment; variants of a program appear as alternative nodes at the same chain slot. Nodes are augmented with their summaries/observations from the last step.

Finally, Hydra reads the query parameters to determine the subgraph to which it applies, recovers relevant summaries, and then executes an SMT query to combine relevant summaries and compute whether the assumption holds across all paths. Violations are reported with the SMT model as a concrete chain input — packet bytes plus any initial map state — that the operator can replay to reproduce the violation locally.

Hydra's main differences from traditional compositional symbolic execution are the BPF queries and relevant observations (§3.2), how the BPF program graph is reconstructed in SMT with stitched summaries, as well as the exploration-pruning strategy (§3.3) and our enhancement of helper models (§3.4).

Query	Description
No Overwrites	Given a shared field (e.g., `skb->mark`): is there an input on which a preceding program's write to that field gets overwritten by a later program?
Parallel Invariance	Given two variants of the same program deployed at the same slot across two chains: is there an input on which the variants return different verdicts?
Invariant Execution	Given a must-execute program D: is there an input where a predecessor terminates execution before D runs?

Table 1: Example Hydra queries. Each is checked against every chain the deployment permits.

3.2 Symbex Instrumentation

Query-specific observables. For each query class in Table 1, Hydra compiles the per-program harness so that the SEE emits, on every path, the observations the query consumes: a write flag for the shared field given to *No Overwrites*, the returned verdict for *Parallel Invariance* and *Invariant Execution*, and the path condition in all cases.

Per-program symbex. Instrumented programs are symbolically executed in isolation. The resulting per-path records are attached to nodes of the deployment graph from §3.1 — each graph node now carries a set of per-path summaries which Hydra will later stitch for chain-level verification using SMT.

3.3 Merging and Fast Verification

Once each program in the deployment carries the per-path summary from §3.2, the question Hydra hands to the SMT solver is a single existential: *is there an input that, when given to any of the possible chains, will violate the queried property?* If yes, the solver returns the input concretely — a packet (and any initial map state) the operator can replay to reproduce the violation locally. If no, the property holds for every input on every chain the graph permits.

Stitching – Shared input glues programs together. Each per-program summary describes its paths as conditions over a symbolic input. Hydra makes a simplification: when two programs run in a chain, they operate on the same input. So if Hydra asks *"does there exist a packet that reaches path i of program A **and** path j of program B?"*, the SMT solver only needs to satisfy the conjunction of both programs' path conditions over the same symbolic `input`. If the solver finds a satisfying assignment, that assignment is a real, end-to-end chain input.

Each query adds its own predicate for violation. Stitching is the substrate; each query layers what counts as violation on top. *No Overwrites* adds: both stitched paths must write the same queried field (thus the second overwrites the first). *Parallel Invariance* adds: the two stitched variants must disagree on verdict. *Invariant Execution* adds: some predecessor of the must-execute program early-terminates.

The pattern generalizes: any cross-program property expressible as "some combination of per-path observables triggers it" fits the same substrate. Adding a new query class requires only (i) telling the harness which per-path observation to record (§3.2) and (ii) layering its violation predicate on the stitched-edge formula.

Soundness of the stitch. Stitching glues programs over one shared `input`, but in a real chain each program runs on the state its predecessors *wrote*: if A overwrites a field that B later branches on, B may follow a path the stitch never considered. Instead of simulating those writes (future work; § 6), Hydra checks that they cannot matter. Statically, from the pruning slice (§ 3.3), it computes for each program the fields it may *write* and the fields that *steer* it (e.g., variables inside `if` statements, or variables that lead to its verdict). A stitched chain can be statically deemed *faithful* despite being write-agnostic when no program is steered by a field an earlier program may have written. On a faithful chain the shared-`input` stitch yields exactly what the real, write-carrying execution would, so the verdict and counterexample Hydra reports are exact. Hydra refuses to process chains it cannot statically verify that its decomposition would be sound on.

Optimization: Query-specific Pruning. Real-code BPF programs produce tens of thousands of paths under naive symbex, the majority of which cannot affect the queried observation. Hydra augments the SEE's search with pruning: essentially, we early-terminate paths which are moot (e.g., for a query about overwriting a variable, we can early terminate any SEE subtree where writes to the variable are no longer reachable). For this, we compute a backwards reachability slice over each program's control flow graph and mark for pruning all basic blocks that cannot transitively reach the queried observation (e.g., the write statement). The SEE then symbolically executes the pruned bitcode, producing only the paths that were relevant to evaluate the query to begin with.

Optimization: Summary Caching. In real deployments, the same set of programs is queried many times as the deployed graph evolves. Hydra exploits this by symbolically executing each (program,version) pair once and caching its summary; subsequent queries re-use the cache instead of re-running the SEE.

3.4 Faithful BPF Helper Models

Up until now we have discussed how to make Hydra scalable and efficient. Now we shift into how Hydra improves the helper models that are critical to symbolically execute eBPF programs.

LLMs are now capable of producing more thorough BPF helper models than the `ebpf-se` ones (Fig. 1b): from verbatim kernel source they can extract more error-return paths, more context-mutation side effects, and a finer-grained match to the kernel's actual behavior. The problem is hallucination — an LLM-generated model may include paths the kernel does not exhibit, or omit paths it does. Our insight is to guardrail the LLM with some notion of ground-truth. In Hydra, we add testing and kernel tracing to the LLM loop.

Pipeline. For each helper, we drive an LLM to produce a model from the kernel source. For every path the model emits (i.e., when symbexed), we synthesize a concrete input and replay it through the actual running kernel via `BPF_PROG_TEST_RUN`, with `kcov` capturing the per-input basic-block fingerprint of the helper's execution. Hydra commits to the model-emitted paths whose kernel fingerprint matches between LLM and kernel; the remaining (unmatched) paths are flagged for the LLM's next refinement iteration and not treated as confirmed downstream. This allows us to prune several hallucinations.

4 Evaluation

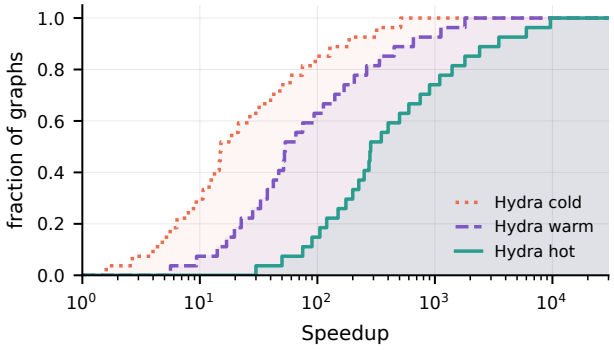


Figure 3: Hydra Speedup over Monolithic KLEE

Setup. All experiments run on a 24-core Intel desktop with 64 GB RAM, KLEE 3.3 / LLVM 16, Z3, and a kcov-enabled Linux kernel v6.14. The program corpus (Table 2) includes open-source BPF programs (e.g., DINT, Crab) and production systems (e.g., Mizar, Katran) plus a set of micro-benchmarks which are symbex-heavy. Chains are assembled by sampling from the corpus at varying depth (d programs) and per-program variants (b versions). We report Hydra in three deployment scenarios: *cold* (every chain re-symbexes every program), *warm* (each program symbexed on the critical path of Hydra only once per deployment graph, then summaries cached and reused), and *hot* (per-program summaries pre-computed by the organization's CI/CD outside the verification critical path of Hydra). Monolithic KLEE has no analogous reuse and is always cold.

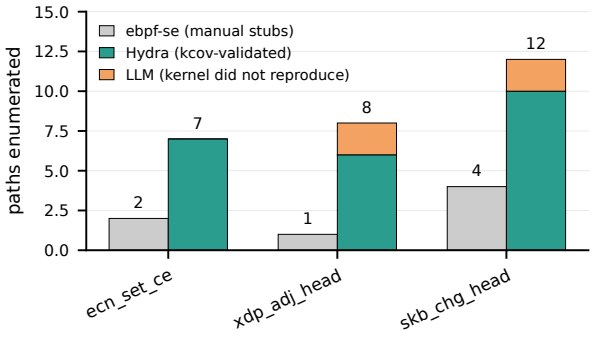


Figure 4: Helper coverage. For each helper, `ebpf-se`'s manually developed stub vs. the LLM-loop model: the LLM bar's bottom segment is what `kcov` validates against the running kernel ("Hydra"); the top is what the test harness did not reproduce.

Programs from Open Source Github Repositories					
`katran` [18]	2,040	`mizar-tr.` [6]	1,905	`mizar-off.` [6]	1,092
`fw` [4]	185	`DINT` [27]	211–558	`ruport` [25]	352
`fluvia` [20]	268	`crab` [14]	579	`dae` [8]	2,430
Micro-benchmark Programs					
`csum` [3]	15	`hash` [23]	15	`compress` [10]	26

Table 2: Hydra program corpus (LoC).

Scaling. Fig. 3 reports the distributions of Hydra speedup over monolith across our entire corpus (27 randomly generated graphs with depth $d \in \{3, 5, 7\}$ and per-slot variants $b \in \{1, 2, 4\}$ – far below complexity estimates from Meta in § 1). The graphs are, by design, disjoint in read-writes — hence every chain is *faithful* (§ 3.3), where the shared-`input` stitch is provably exact. We observe that even for the cold scenario, Hydra's pushdown and per-program symbex amortization yield a median speedup of ∼ 15× and an aggregate speedup of ∼ 80×. In the warm scenario, where each program is symbexed once and cached for reuse, the median speedup rises to ∼ 53× with an aggregate of ∼ 282×. Finally, in the hot scenario, where per-program summaries are pre-computed, we achieve a median speedup of 284× and an aggregate of 1,506×. These results demonstrate that Hydra effectively mitigates the state explosion problem inherent in monolithic analysis. Accordingly, Hydra and a write-aware monolithic baseline agree on every verdict (violations and non-violations alike).

Helper coverage. Fig. 4 compares `ebpf-se`'s manually developed stubs to the LLM-loop model on three representative helpers. The LLM-loop enumerates 27 paths total against `ebpf-se`'s 7 (2.5–6× per helper). Of those 27, **23** are kcov-validated against the running kernel and form Hydra's helper model; the remaining 4 are unconfirmed: the `BPF_PROG_TEST_RUN` harness neither reproduces nor refutes them, so each is either a model hallucination or a real kernel path the harness cannot yet drive.

5 Related Work

In-Kernel eBPF Verification and Static Analysis. The foundational security mechanism for eBPF is the Linux kernel's verifier, which ensures that programs are memory-safe, appropriately bounded, and guaranteed to terminate before being loaded [22]. Similarly to the verifier, eBPF-based static analysis tools (e.g., Yaksha [21]) focus on source-level dataflow to assert properties; These works, however, cannot introspect on the many possible execution paths individually, and will produce answers at a granularity too coarse: e.g., if any codepath (even one impossible at runtime) might create a property violation (e.g., metadata overwrite), the entire program fails. Our intention with Hydra is to have more nuanced chain verification that is closely aligned with how unit tests are used to validate specific interactions.

Symbolic Execution for eBPF. To achieve deeper semantic analysis, recent tools have applied symbolic execution to the eBPF domain. Systems like DRACO [17] utilize KLEE to perform functional validation. However, DRACO is designed for single-program verification. When forced to analyze many different program chains it suffers from path and state explosion.

Multi-Program BPF Optimization. Similarly to Hydra, modeling eBPF deployments at a granularity higher than individual programs (e.g., chains or pairs of programs) has allowed eBPF optimization work [7, 15] to achieve significant performance gains that would be impractical to achieve with single-program optimization alone.

Dataplane and NFV Composition Verification. Verification of network function chaining (NFV) and programmable dataplanes has been extensively studied. Frameworks such as P4v [16], Vigor [26], successfully verify end-to-end properties across sequences of network components.

6 Limitations and Future Work

Write propagation. Hydra answers only chains whose stitch is provably exact (§ 3.3) and defers the rest — those an upstream write may steer (e.g. through a shared map); covering write-aware chains is our immediate future work.

Automated Query Compilation. Queries are manually developed SMT formulas plus manually developed instrumentation/optimization. A unifying future-work item is a query engine that takes a developer-facing specification (e.g., "`skb->mark` written by prog *A* must persist until the kernel observes it") and compiles both the optimized SMT formula and the matching pushdown targets. We believe this is within reach given that our family of queries and optimizations have the same underlying structure.

`BPF_PROG_TEST_RUN` does not perfectly reproduce kernel behavior. `BPF_PROG_TEST_RUN` is used as Hydra's Oracle in its helper-modeling pipeline; Such tool does not reproduce the kernel's full execution environment one-to-one (e.g., no skb clones), so a non-trivial fraction of paths the kernel actually exhibits cannot be reproduced at validation time.

7 Conclusion

We presented Hydra: a verifier that lifts eBPF verification from individual programs to deployment graphs, and optimizes both the scalability (with intra- and inter-program optimizations) and soundness (with a kcov-grounded, LLM-based helper-modeling pipeline). Our evaluation shows that Hydra verifies cross-program properties on real production source — katran, mizar, and similar deployments — while avoiding the state explosion inherent to monolithic analysis, and recovers helper-model paths that manually developed models often miss.

Acknowledgments

We thank the eBPF '26 reviewers for their thoughtful feedback. We also thank Prankur Gupta and Sebastiano Miano for insightful discussions.

References

[1] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. 2007. *Demand-Driven Compositional Symbolic Execution.* Technical Report MSR-TR-2007-138. 19 pages. https://www.microsoft.com/en-us/research/publication/demand-driven-compositional-symbolic-execution/

[2] Theophilus A. Benson and Prankur Gupta. 2025. Fast and the Furious: CICD Pipeline for eBPF Programs at Meta Scale. KubeCon + CloudNativeCon North America, https://kccncna2025.sched.com/event/27FeA.

[3] R. Braden, D. Borman, and C. Partridge. 1988. Computing the Internet Checksum. RFC 1071.

[4] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. 2020. hXDP: Efficient Software Packet Processing on FPGA NICs. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 973–990. https://www.usenix.org/conference/osdi20/presentation/brunella

[5] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209–224.

[6] Centaurus Infrastructure. 2026. *Mizar*. https://github.com/CentaurusInfra/mizar Accessed June 2026.

[7] Milo Craun and Dan Williams. 2025. Pairwise BPF Programs Should Be Optimized Together. In *Proceedings of the 3rd Workshop on EBPF and Kernel Extensions (eBPF '25)*. Association for Computing Machinery, New York, NY, USA, 22–27. https://doi.org/10.1145/3748355.3748362

[8] daeuniverse. 2026. dae: eBPF-based transparent proxy. https://github.com/daeuniverse/dae. Accessed August 2026.

[9] Dependable Systems Laboratory (DSLab), EPFL. 2022. eBPF-SE: Tool to symbolically execute eBPF programs. https://github.com/dslab-epfl/ebpf-se.

[10] Sven Fiergolla and Petra Wolf. 2021. Improving Run Length Encoding by Preprocessing. *CoRR* abs/2101.05329 (2021). arXiv:2101.05329 https://arxiv.org/abs/2101.05329

[11] Patrice Godefroid. 2007. Compositional dynamic test generation. *SIGPLAN Not.* 42, 1 (Jan. 2007), 47–54. https://doi.org/10.1145/1190215.1190226

[12] Prankur Gupta, Takshak Chahande, Miao Xu, Prashanth Kannan, Tejas Birajdar, Lilian Liu, Rajiv Krishnamurthy, and Theophilus A. Benson. 2026. BPFComposer: Towards deterministic and safe composition of eBPF Programs (and their Shared state) *(DEBS '26)*. Association for Computing Machinery, 150–156. https://doi.org/10.1145/3809481.3812620

[13] Rishabh Iyer, Katerina Argyraki, and George Candea. 2022. Performance Interfaces for Network Functions. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 567–584. https://www.usenix.org/conference/nsdi22/presentation/iyer

[14] Marios Kogias, Rishabh Iyer, and Edouard Bugnion. 2020. Bypassing the load balancer without regrets. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC '20)*. Association for Computing Machinery, 193–207. https://doi.org/10.1145/3419111.3421304

[15] Hsuan-Chi Kuo, Kai-Hsun Chen, Yicheng Lu, Dan Williams, Sibin Mohan, and Tianyin Xu. 2022. Verified programs can party: optimizing kernel extensions via post-verification merging. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 283–299. https://doi.org/10.1145/3492321.3519562

[16] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. 2018. p4v: practical verification for programmable data planes. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 490–503. https://doi.org/10.1145/3230543.3230582

[17] Dana Lu, Boxuan Tang, Michael Paper, and Marios Kogias. 2024. Towards Functional Verification of eBPF Programs. In *Proceedings of the ACM SIGCOMM 2024 Workshop on EBPF and Kernel Extensions* (Sydney, NSW, Australia) *(eBPF '24)*. Association for Computing Machinery, New York, NY, USA, 37–43. https://doi.org/10.1145/3672197.3673435

[18] Meta Platforms, Inc. 2018. Katran: A high performance layer 4 load balancer. https://github.com/facebookincubator/katran. Accessed: August 31, 2026.

[19] Netflix Technology Blog. 2025. How Netflix Accurately Attributes eBPF Flow Logs. https://netflixtechblog.com/how-netflix-accurately-attributes-ebpf-flow-logs-afe6d644a3bc. Accessed August 2026.

[20] NTT Communications. 2026. Fluvia: IPFIX exporter using eBPF. https://github.com/nttcom/fluvia. Accessed August 2026.

[21] Animesh Singh, K Shiv Kumar, S. VenkataKeerthy, Pragna Mamidipaka, R V B R N Aaseesh, Sayandeep Sen, Palanivel Kodeswaran, Theophilus A. Benson, Ramakrishna Upadrasta, and Praveen Tammana. 2026. Yaksha-Prashna: Understanding eBPF Bytecode Network Function Behavior. arXiv:2602.11232 [cs.CR] https://arxiv.org/abs/2602.11232

[22] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. 2023. Verifying the Verifier: eBPF Range Analysis Verification. In *Computer Aided Verification: 35th International Conference, CAV 2023, Paris, France, July 17–22, 2023, Proceedings, Part III* (Paris, France). Springer-Verlag, Berlin, Heidelberg, 226–251. https://doi.org/10.1007/978-3-031-37709-9_12

[23] Wikipedia contributors. 2026. Toeplitz Hash Algorithm. https://en.wikipedia.org/wiki/Toeplitz_Hash_Algorithm. Accessed August 2026.

[24] Qiongwen Xu, Michael D. Wong, Tanvi Wagle, Srinivas Narayana, and Anirudh Sivaraman. 2021. Synthesizing safe and efficient kernel extensions for packet processing. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 50–64.

[25] xxxxnnxxxx. 2026. ruport. https://github.com/xxxxnnxxxx/ruport. Accessed August 2026.

[26] Arseniy Zaostrovnykh, Solal Pirelli, Rishabh Iyer, Matteo Rizzo, Luis Pedrosa, Katerina Argyraki, and George Candea. 2019. Verifying software network functions with no verification expertise. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 275–290.

[27] Yang Zhou, Xingyu Xiang, Matthew Kiley, Sowmya Dharanipragada, and Minlan Yu. 2024. DINT: Fast In-Kernel Distributed Transactions with eBPF. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 401–417. https://www.usenix.org/conference/nsdi24/presentation/zhouyang