Toward Safely Optimizing eBPF Programs using Transformation Witnesses

Xiangyu Gao[w] Alex de Cock Buning[c] Xiangfeng Zhu[w] Jocelyn Qiaochu Chen[a]
Konstantinos Kallas[u] Arvind Krishnamurthy[w] Ratul Mahajan[w]
[w] *University of Washington* [c] *Carlos III University of Madrid* [a] *University of Alberta* [u] *UCLA*

Abstract

The execution speed of eBPF programs is important because they implement many performance-critical functions. But existing eBPF optimizers are limited because they primarily operate on bytecode and make local optimizations, missing non-local optimization opportunities visible at the source-code level. We present EvolveBPF, a framework that uses AI agents for source-level eBPF optimizations. Different from the existing frameworks, it requires the AI agent to generate not only optimized eBPF programs but also *transformation witnesses* that capture the assumptions under which the code optimization is correct. We incorporate these witnesses into a semantic checker, enabling code optimizations to go beyond what would be possible with strict equivalence alone. Experiments using open-source benchmarks shows that our approach can speed up eBPF programs by up to 2.7x.

CCS Concepts

• **Software and its engineering → Source code generation**.

Keywords

eBPF; Transformation Witness; Code Optimization; Semantic Check; Coding Agent

ACM Reference Format:
Xiangyu Gao, Alejandro de Cock Buning, Xiangfeng Zhu, Jocelyn Qiaochu Chen, Konstantinos Kallas, Arvind Krishnamurthy, Ratul Mahajan. 2026. Toward Safely Optimizing eBPF Programs using Transformation Witnesses. In *The 25th ACM Workshop on Hot Topics in Networks (HotNets '26), November 16–17, 2026, Salt Lake City, UT, USA*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3847290.3859947



This work is licensed under a Creative Commons Attribution 4.0 International License.
HotNets '26, Salt Lake City, UT, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-3029-0/2026/11
https://doi.org/10.1145/3847290.3859947

1 Introduction

eBPF offers a flexible approach to safely extend modern operating systems without modifying the kernel. A wide range of applications use eBPF for packet processing [14], load balancing [17, 19], observability [13, 20], access control, etc. eBPF is also expanding into new domains such as user-space instrumentation [25] and GPU profiling [23], extending its applicability beyond the Linux kernel.

The efficiency of eBPF programs is critical because they are often invoked on every packet, system call, or kernel event. For instance, Meta's Katran [7] is an eBPF-based load balancer that processes every incoming packet, making application performance depend intimately on its runtime.

The research community has proposed two categories of techniques to optimize eBPF applications. The first relies on compiler optimizations [16, 21] and does local, bytecode-level transformations (e.g., eliminating redundant instructions). While helpful, the optimization space of these techniques is fundamentally constrained by the information retained in the bytecode. For instance, they cannot reduce the number of predicate evaluations by eliminating redundant checks. The second category, exemplified by systems such as Cilium [3], optimizes runtime parameters and system settings. These techniques do not optimize eBPF programs themselves.

We explore using AI agents [1, 2, 8] for eBPF program optimization. These agents have demonstrated strong capabilities for optimizing code for a wide range of domains (e.g., C, python, Rust). But using them requires addressing the unreliability of AI-generated transformations as well as the unique constraints of eBPF programs. An AI agent may produce an incorrect program. Developers commonly mitigate this risk by testing the original and optimized programs on a set of test cases, but this approach is not foolproof. As an alternative, one might use a formal equivalence checker such as Heimdall [10]. Such checking offer strong correctness guarantee but it tends to be too restrictive. Specifically, AI agents can propose a broad spectrum of transformations (§3), ranging from operation level edits to new data structures and representations. Equivalence checkers can often not certify the equivalence of such transformations, and excluding them sacrifices valuable optimization opportunities.

These limitations motivate EvolveBPF, our framework for source-level eBPF optimization (§3.1). Like iterative optimization frameworks like OpenEvolve [8], EvolveBPF repeatedly prompts an AI agent to generate optimized eBPF candidate programs. In addition, we prompt it to also generate a *transformation witness* that captures conditions under which the code transformation is correct. Transformation witnesses have two components: (i) assumptions which specify the conditions that the agent relies on for the optimization (e.g., the value of certain variable cannot exceed an inferred constant)—the validity of these assumptions must be confirmed by developers; and (ii) bindings which specify correspondences between variables in the original and optimized programs. Transformation witnesses enable EvolveBPF's semantic checker to certify transformations that conventional equivalence checkers cannot handle.

We evaluate EvolveBPF (§4) on eBPF programs from the open-source BCC `libbpf-tools` [6]. EvolveBPF supports two optimization modes: *naive mode* requires strict semantic equivalence, and *aggressive mode*, which permits transformation-witness-aware transformations. Both optimization modes improve the execution performance across all benchmarks. The aggressive mode achieves an average speedup of 1.9× (up to 2.7×), outperforming the naive mode, which achieves an average speedup of 1.2× (up to 1.6×).

Even though EvolveBPF targets eBPF programs, its key insight of transformation-witness-guided code optimization can be applied to other domains (e.g., GPU kernels, network functions, and databases) where a semantic checker can be built. We therefore see EvolveBPF as an initial step toward similar systems for other such domains.

2 Background and Related Work

eBPF has become fundamental for extending the Linux and Windows kernels at runtime for many applications. We make four observations about the current eBPF landscape.

Observation 1: eBPF use is increasing across modern systems. eBPF provides a lightweight and verifiably safe mechanism for extending the kernel without modifying its source. Developers have adopted eBPF for a growing range of kernel functionality, including packet processing [14], policy extension [26, 27], observability [23], and network protocol deployment [9, 18].

Although eBPF was initially designed as a mechanism for extending the kernel, its use has expanded well beyond kernel extensions. User-space eBPF runtimes and frameworks enable developers to execute eBPF programs in application-level environments [25]. At the same time, eBPF has evolved from a tracing-oriented technology into a general-purpose programmable systems platform, powering networking, observability, and security services at production scale [3, 7].

Observation 2: eBPF performance has system-wide impact. eBPF programs are typically compiled into eBPF bytecode and executed inside the kernel where they participate in fundamental system operations (e.g., scheduling, storage I/O, system call handling). eBPF programs thus frequently execute on performance-critical paths, where even small reductions in execution latency are amplified across millions of invocations. Kernel developers have long applied techniques such as aggressive function inlining to keep these paths fast; the same level of attention should be paid to eBPF programs.

Moreover, many eBPF programs are deployed for observability, tracing, or security monitoring, where their purpose is to observe the system without perturbing it. For these workloads, minimizing runtime overhead is essential.

Observation 3: Checking the equivalence of two eBPF programs is possible. Preserving program semantics is essential for code optimization. Formally proving the equivalence using formal methods offer better guarantee than testing-based approaches. However, determining whether two arbitrary programs are semantically equivalent is undecidable in general [5]. Fortunately, eBPF has a restricted execution model that disallows unbounded loops and recursion. These unique properties offer opportunities to formally verify the semantic equivalence between eBPF programs. For instance, Heimdall [10] generates Rust-style eBPF programs from C-style ones and formally verifies them using symbolic execution.

Observation 4: eBPF optimization opportunities go beyond strict semantic equivalence. Traditional code optimizers operate within the space of strict semantic equivalence: typical examples include replacing expensive operators with more efficient ones (e.g., replacing multiplication by powers of two with bit shifts) and simplifying conditional branches. Such optimizers have been developed for eBPF as well (e.g., K2 [21] and Merlin [16]). While such optimizations remain applicable to eBPF programs, they represent only a fraction of the practical optimization opportunities. Performance is also heavily influenced by program-level design choices [4], and many improvements arise from redesigning program structures or data representations. For example, an eBPF program that only requires a file's basename may allocate a 256-byte buffer instead of storing the entire 4096-byte pathname. Such a transformation changes the program's memory layout and observable internal state, yet still satisfies the application's intended functionality.

Thus, Improving the execution efficiency of eBPF programs is important and the restricted nature of eBPF makes it possible to do so with strong correctness guarantees, but current optimizers use traditional methods that are limited in what they can accomplish.

AI agents [8, 15, 22] offer a promising way to discover a broader range of optimization opportunities that go beyond

strict semantic equivalence. Prior work (e.g., SimpleBPF [11] and KGent [24]) has used LLMs to generate eBPF programs but offers no correctness guarantees. These observations motivate our framework for safe, agentic eBPF optimization.

3 EvolveBPF Design

EvolveBPF is an agent-guided framework for safe and effective eBPF optimization (Figure 1). It iteratively asks an AI agent to generate optimized candidates and transformation witnesses of an eBPF program. It then validates each candidate for semantic equivalence based on those witnesses, BPF verifier acceptance, and execution performance.

EvolveBPF follows OpenEvolve's evolutionary paradigm, iterating to progressively accumulate optimization knowledge. Given an initial eBPF program and a user-specified iteration budget, EvolveBPF repeatedly invokes the AI agent to generate candidate programs, each of which is evaluated by the validation pipeline. EvolveBPF records the evaluation results together with the optimization rationale and feeds them back into the next iteration, enabling the agent to refine previous transformations and explore new strategies. This feedback loop continues until the iteration budget is exhausted, at which point EvolveBPF returns the best-performing validated program.

3.1 Optimizations

We analyze the eBPF optimizations generated by the agent and group them into three categories: (1) assumption-dependent optimizations, whose correctness depends on developer-confirmed assumptions about the target execution environment; (2) representation-level optimizations, which preserve program semantics while redesigning data type representations; and (3) operation-level optimizations, which rewrite local operations without changing the program structure. Of these, only operation-level optimizations are supported by traditional compiler frameworks, so the rest of this section focuses on the first two categories, which are uniquely enabled by EvolveBPF.

3.1.1 Assumption-Dependent Optimization. This category comprises transformations whose correctness depends on assumptions about the target execution environment. Unlike conventional compiler optimizations, which must hold for all executions, these transformations are valid only when their underlying assumptions hold.

Figure 2 illustrates one such optimization discovered by the AI agent. The original program (Figure 2(a)) monitors file read and write operations and maintains per-file statistics in a hash map. When a file is first observed, the program allocates a new map entry, records the process information, and stores the file name in the `filename` field. Subsequent accesses update the corresponding read/write counters without modifying the stored metadata.

The original implementation allocates a `PATH_MAX` (4096-byte) char array to store `filename`. However, examining the implementation of `get_file_path()` reveals that the program only reads `dentry→d_name.name`, which corresponds to the file's basename rather than its absolute pathname. Since Linux limits filename length to 255 characters, the AI agent replaces the buffer with a 256-byte array and copies the filename using `bpf_probe_read_kernel_str()` (Figure 2(b)), significantly shrinking every hash-map entry.

This optimization is not universally applicable: its correctness depends on properties of the target execution environment, such as the `NAME_MAX` limit. Such optimizations cannot be derived from the program alone, as they require reasoning about operating-system semantics beyond the program source. **EvolveBPF therefore requires the AI agent to emit the underlying assumptions as transformation witnesses, which developers can verify against their deployment environment.**

3.1.2 Representation Optimization. Representation-level optimizations preserve the semantics of an eBPF program while redesigning its internal data representations. Rather than modifying individual operations, these optimizations reorganize data structures, memory layouts, or map usage to reduce execution overhead. Such transformations typically require reasoning about the relationships among multiple program components, placing them beyond the reach of traditional compiler passes.

Figure 3 shows a representative optimization discovered by EvolveBPF. The original XDP program counts incoming packets, with a control flag that determines whether counting is enabled. To maintain these two pieces of state, the program defines two independent BPF maps: `ctl_array`, which stores the control flag, and `cntrs_array`, which stores the packet counter. During packet processing, the program first looks up `ctl_array` to check whether counting is enabled and, if so, performs a second lookup on `cntrs_array` to update the counter. Consequently, every packet incurs two BPF map lookups.

The AI agent observes that both maps are indexed by the same key and are always accessed together. It therefore merges them into a single map whose value is a composite structure containing both the control flag and the packet counter. As a result, the program performs only one map lookup per packet while preserving the original semantics. Since map lookups are among the most expensive operations in eBPF programs, eliminating one lookup significantly reduces the overhead of the packet-processing path.

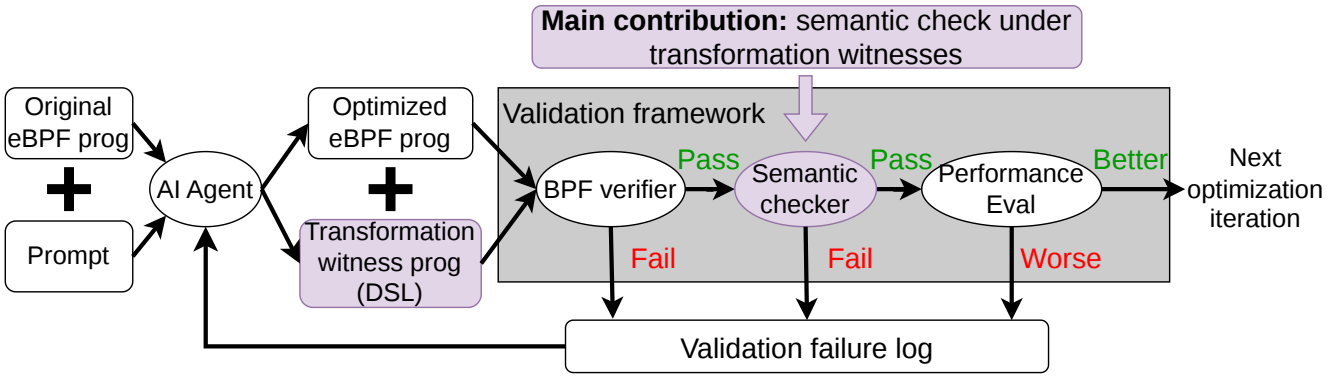


Figure 1: Given an input eBPF program and prompts, the AI agent generates candidate programs and transformation witnesses. Each candidate is evaluated from three perspectives: BPF verifier compliance, transformation-witness-guided semantic check, and execution performance evaluation. The evaluation results are fed back to the AI agent to guide the next iteration. This optimization loop continues until the iteration threshold.

```
#define PATH_MAX 4096
struct file_stat {
    ...
    char filename[PATH_MAX];
};
static void get_file_path(...) {
    struct qstr dname;
    dname = BPF_CORE_READ(...);
    bpf_probe_read_kernel(...);
}
static int probe_entry(...) {
    struct file_stat *valuep;
    ...
    get_file_path(file, valuep->filename,
                  sizeof(valuep->filename));
}
```

(a)

```
#define FILE_NAME_LEN 256
struct file_stat {
    ...
    char filename[FILE_NAME_LEN];
};

static int probe_entry(...) {
    struct file_stat *valuep;
    ...
    bpf_probe_read_kernel_str(valuep->filename,
          sizeof(valuep->filename), name);
}
```

(b)

Figure 2: (a) is the original eBPF program while (b) is an optimized version by reducing the size of filename from 4096 bytes to 256 bytes.

Although the optimized program is semantically equivalent to the original, conventional equivalence checkers generally fail to verify this transformation because the underlying data representations are no longer identical. The original program stores the control flag and packet counter in two separate maps, whereas the optimized version stores them in a single composite object. Establishing equivalence therefore

```
struct {
  __type(key, __u32); __type(value, __u32);
  ...} ctl_array SEC(".maps");
struct {
  __type(key, __u32); __type(value, __u64);
  ...} cntrs_array SEC(".maps");
int pktcntr(struct xdp_md* ctx) {
  __u32 ctl_flag_pos = 0; __u32 cntr_pos = 0;
  __u32* flag = bpf_map_lookup_elem(&ctl_array,
                                    &ctl_flag_pos);
  if (!flag || (*flag == 0)) {...};
  __u64* cntr_val = bpf_map_lookup_elem(&cntrs_array,
                                        &cntr_pos);
  if (cntr_val) {...};
}
```

(a)

```
struct datarec {__u32 enabled;__u64 counter;};
struct {
  __type(key, __u32); __type(value, struct datarec);
  ...
} stats_map SEC(".maps");
int pktcntr(struct xdp_md* ctx) {
  __u32 key = 0;
  struct datarec *rec = bpf_map_lookup_elem(&stats_map,
                                            &key);
  if (!rec || rec->enabled == 0){...}
  ...
}
```

(b)

Figure 3: (a) is the original eBPF program defining 2 maps while (b) is an optimized version by merging them into one map.

requires reasoning about the correspondence between the two data representations.

3.2 Semantic Checker under Transformation Witnesses

Step 2 of the validation pipeline in Figure 1 performs semantic validation using developer-approved transformation witnesses. We next use the example in Figure 2 to illustrate how these witnesses are generated, formalized, and used to establish semantic equivalence.

```
⟨program⟩ ::= ⟨assumption-block⟩ ⟨binding-block⟩
⟨assumption-block⟩ ::= assumption { ⟨assumption-list⟩ }
⟨assumption-list⟩ ::= ⟨assumption-statement⟩
                    | ⟨assumption-statement⟩ ⟨assumption-list⟩
⟨assumption-statement⟩ ::= ⟨range-assumption⟩ ;
⟨range-assumption⟩ ::= ⟨original-expression⟩ in [ ⟨integer⟩ , ⟨integer⟩ ]
⟨binding-block⟩ ::= binding { ⟨binding-list⟩ }
⟨binding-list⟩ ::= ⟨binding-statement⟩
                 | ⟨binding-statement⟩ ⟨binding-list⟩
⟨binding-statement⟩ ::= ⟨original-expression⟩ = ⟨optimized-expression⟩ ;
⟨original-expression⟩ ::= original.⟨expression⟩
⟨optimized-expression⟩ ::= optimized.⟨expression⟩
⟨expression⟩ ::= ⟨variable⟩ | ⟨map-expression⟩ | ⟨field-expression⟩
⟨map-expression⟩ ::= ⟨variable⟩ [⟨integer⟩ :⟨integer⟩ ] | ⟨variable⟩ [⟨variable⟩]
⟨field-expression⟩ ::= ⟨map-expression⟩ .⟨field-name⟩ | ⟨variable⟩ .⟨field-name⟩
⟨variable⟩ ::= ⟨identifier⟩
⟨field-name⟩ ::= ⟨identifier⟩
⟨integer⟩ ::= ⟨digit⟩ | ⟨digit⟩ ⟨integer⟩
⟨identifier⟩ ::= ⟨letter⟩ | ⟨letter⟩ ⟨identifier-tail⟩
⟨identifier-tail⟩ ::= ⟨letter⟩ | ⟨digit⟩ | _ | ⟨letter⟩ ⟨identifier-tail⟩
                    | ⟨digit⟩ ⟨identifier-tail⟩ | _ ⟨identifier-tail⟩
⟨letter⟩ ::= a | ··· | z | A | ··· | Z
⟨digit⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Figure 4: BNF grammar for the transformation-witness DSL.

DSL design for transformation Witness. We want transformation witnesses to satisfy two main goals: (1) easy for human developers to understand and review, and (2) precise enough for automated semantic checking. As a result, Figure 5 shows an example where we output both the transformation reason in natural language and formally describe transformation witnesses using the designed DSL.

The grammar design for the transformation witness DSL is shown in Figure 4. A DSL program consists of 2 main parts: **assumption** and **binding**. The assumption portion shows necessary assumptions made to support the program optimization. All assumptions in this component must be reviewed and approved by human developers. The binding portion relates variables in the original and optimized programs. The subsequent semantic checker can verify equivalence according to all variables' binding relationships.

For example, Figure 5 illustrates one DSL program. The Reason part provides a human-readable explanation of the transformation witness for developers to review. Then, the assumption portion shows that only 256 out of 4096 bytes in the filename variable contain meaningful data. Therefore, the generated optimized program only contains a 256-byte filename variable and the binding component establishes the correspondence between this variable and the first 256 bytes of the filename variable in the original program.

Transformation Witness Generation. EvolveBPF prompts the AI agent to produce structured transformation witnesses that specify the conditions under which an optimization is correct. Each witness pairs the agent's natural-language reasoning with a formal representation. For instance, Figure 5 illustrates the witnesses for the filename representation optimization. The AI agent first explains the rationale behind the optimization (e.g., the optimized program stores only the basename, whose size is at most 256 bytes); EvolveBPF then formalizes this rationale into an executable assumption that the semantic checker can directly consume. Presenting both forms enables developers to easily inspect and validate the underlying reasoning: compared with manually establishing the semantic equivalence of two complete programs, validating a small set of explicit witnesses requires less effort.

Our witness design echoes recent formal verification systems such as TorchLean [12], which bridges the gap between model execution and verification by making implicit semantics (e.g., preprocessing and floating-point behavior) explicit and machine-checkable.

Semantic Checker. Different from conventional semantic checker, EvolveBPF performs equivalence checking under transformation witnesses. After developers approve the assumptions specified in the witness, we use a symbolic execution engine to checks the equivalence between the original and optimized programs' output, using the binding relationships to reconcile variables whose names or types have changed during optimization.

For the example in Figure 5, the code optimization agent identifies that the program only uses the basename instead of the pathname and assumes that filenames in the target environment are at most 255 bytes long and can fit into a 256-byte buffer, including the null terminator. Although this length limit does not mean that the unused portion of the original 4096-byte buffer is zero-initialized, these bytes' value do not affect the program's observable behavior. Therefore, we can set them to zero without loss of generality, reducing the symbolic input space to the first 256 bytes. Since the optimized program uses only a 256-byte filename buffer, its pre-transform extracts the first 256 bytes of the symbolic input, while its post-transform reconstructs the original 4096-byte representation by zero-padding the remaining bytes. This allows the semantic checker to compare the two programs under a common representation even if some variables' name and type are different.

We build our semantic checker on top of the symbolic verification framework proposed by Heimdall [10], which models registers, stack memory, maps, and helper functions for eBPF programs. For instance, a BPF hash map is modeled as a symbolic Z3 array, while the semantics of BPF helper functions such as `bpf_map_lookup_elem()` are captured by symbolically exploring both cases: the queried key being present or absent.

```
Reason: The program stores only the basename of each
     filename. In Linux, basenames are at most 256 bytes.
Transformation Witness DSL Program:
assumption {
  original.entries[k].filename[256:4096] in [0, 0];
}
binding {
  original.entries[k].filename[0:256] =
      optimized.entries[k].filename;
}
```

Figure 5: Transformation witness generated by the AI agent for the optimization in Figure 2. The assumption constrains the unused filename bytes, while the binding relates the original and optimized representations.

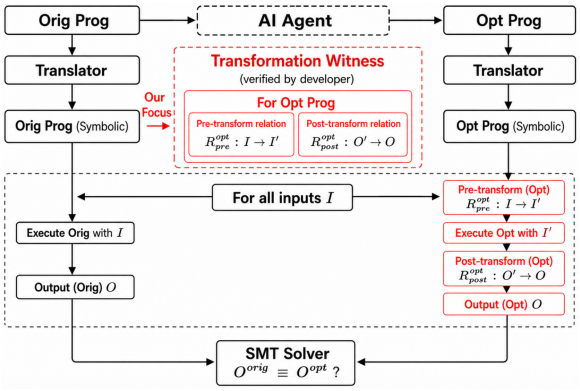


Figure 6: An AI agent generates an optimized program together with a transformation witness consisting of witnesses (verified by the developer), pre- and post-transform relations. The original and optimized programs are symbolically executed, and the SMT solver verifies equivalence between their outputs.

4 Preliminary Evaluation

We evaluate the optimization effectiveness of EvolveBPF by comparing the execution time of the original and optimized eBPF programs. We evaluate EvolveBPF under two optimization modes. The *naive mode* only accepts optimizations that are accepted by a conventional semantic checker. This can be supported by existing optimization frameworks. Our key contribution is the *aggressive mode*, which further enables transformation-witness-guided optimizations that modify the program representation or behavior under developer-approved transformation witnesses. We use the GPT-5 mini model as the AI agent and compare the performance gain achieved by these two modes.

Benchmarks. We select our evaluation benchmarks from the `libbpf-tools` [6] suite in BCC, which contains production-oriented eBPF programs for observing different Linux kernel subsystems. For example, `biostacks` monitors the start and completion of block I/O requests by attaching to block-layer hooks (e.g., `blk_account_io_start`, `blk_account_io_done`, and `block_io_start/done`), while `tcplife` tracks TCP connection lifetimes, transferred bytes, and endpoint information through the `sock_sock_set_state` tracepoint. We compile eBPF programs with Clang, load and attach them to the corresponding kernel hooks using libbpf, and generate requests to repeatedly trigger execution to compare their kernel-side execution time.

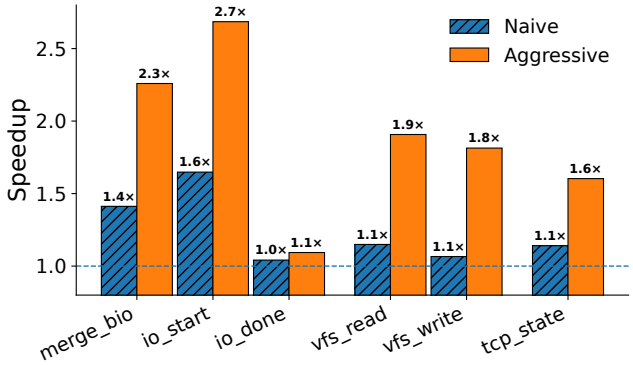


Figure 7: Execution speedup of the optimized eBPF programs generated in naive and aggressive modes over the original implementations across different benchmarks and kernel hooks [6]. The execution time of the original program is normalized to 1.

Results. Figure 7 summarizes the execution speedups achieved by EvolveBPF across 6 representative eBPF programs. Both optimization modes consistently improve performance over the original implementations. The largest speedup (`io_start` hook in `biostacks`) achieves 2.7× execution speedup in the aggressive mode. The root reason is that the optimized program stores only compact request metadata with a stack ID instead of embedding the full stack, batches the gendisk field reads, and uses per-CPU histogram storage to reduce cross-CPU contention during aggregation. These optimizations assume that I/O worker call stacks remain stable over short intervals and user space can aggregate per-CPU histogram values. This assumption is mostly reasonable for database workers and production systems with stable I/O paths.

5 Discussion and Open Problems

Optimizing eBPF programs with AI agents raises several research questions. We address some of them in EvolveBPF, while others remain open and motivate future research.

Q1: What is the appropriate DSL design to describe the transformation witness? EvolveBPF asks the agent to generate both an optimized program and transformation witnesses with the binding between the pre- and post-transformation program states. These AI-generated assumptions require

developers to manually verify their validity. It is therefore important to develop transformation witness representations that are both expressive enough for formal reasoning and intuitive enough for developers to review with minimal effort.

Currently, EvolveBPF uses a DSL to formally specify transformation witnesses. Designing the DSL involves a trade-off between simplicity and expressiveness. A simpler DSL makes it easier for AI agents to understand its syntax and semantics and generate valid witnesses, but may limit the optimization opportunities that can be expressed. For example, our current DSL supports only continuous value ranges and cannot express assumptions that restrict an input variable to several disjoint ranges or specific bit patterns. Extending the DSL to support more expressive witnesses while maintaining its grammar requires more further investigation.

Besides, EvolveBPF relies on AI agents to generate natural-language descriptions to help human developer understand the change and review transformation witnesses. However, there is no formal guarantee that the natural-language descriptions accurately reflect the corresponding DSL programs. Ensuring consistency between these two representations is an important direction for future work.

Q2: What notion of correctness should optimization preserve? EvolveBPF extends traditional semantic equivalence by incorporating AI-generated, developer-approved transformation assumptions into the semantic checker, enabling more practical source-level optimizations (e.g., data representation changes). However, exact semantic equivalence may still be unnecessarily restrictive for many eBPF applications. In practice, developers often care about whether an optimization satisfies design requirements (e.g., both 1s- and 5s- intervals satisfy the profiling requirement). This raises a broader research question: can optimization correctness be defined in terms of application-level objectives rather than exact semantic equivalence? Such a notion would allow AI agents not only to optimize existing implementations, but also to guide developers toward programs that better satisfy their deployment requirements.

Q3: How should the evaluator guide future optimization? EvolveBPF adopts an iterative optimization workflow to let AI agents generate candidate optimizations. In this process, each optimization iteration incurs nontrivial LLM inference time and costs. An important research question is therefore how the evaluator can provide actionable feedback to guide subsequent optimization, reducing unnecessary exploration and improving cost efficiency. This includes developing eBPF-specific prompting strategies, identifying optimization patterns from previous iterations, and generating targeted feedback that helps AI agents converge to high-quality optimizations with fewer iterations.

Q4: Can transformation witnesses generalize beyond eBPF? Transformation witnesses capture a general need in agent-driven program optimization: transformations often rely on domain assumptions that conventional equivalence checkers cannot support. The same idea could potentially apply to domains such as GPU kernels, networking software, and database systems. More broadly, agents could evolve from optimizers into interactive program designers: developers provide an initial implementation and high-level requirements, while agents explore alternative representations and behaviors, with transformation witnesses allowing developers to inspect and validate the resulting design choices. A key open challenge becomes: given a new application domain, how to systematically construct expressive, domain-specific transformation witnesses and integrate them with efficient semantic checkers.

6 Conclusion

Improving eBPF execution efficiency can benefit a wide range of performance-critical systems. We presented EvolveBPF, an agent-guided framework that safely explores source-level optimizations beyond the reach of traditional compiler optimizations via transformation-witness-guided semantic validation. Preliminary results are promising, showing big execution speedup on open-source eBPF benchmarks. We believe the framework can generalize to safe and effective AI-assisted program optimization in broader domains.

Acknowledgments

We thank the HotNets reviewers and our shepherd, Loqman Salamatian, for their feedback. We thank Vishnu Asutosh Dasu for his detailed explanation for the heimdall tool and Longfei Qiu for his help in understanding the symbolic semantic checker. This work was supported by the UW postdoc research award, UW FOCI and its partners (Alibaba, Amazon, Cisco, Google, Microsoft, and VMware), a Canada CIFAR AI Chair award to Jocelyn Qiaochu Chen, and by NSF Grants 2402695, 2525352, 2546642, and CNS-2504469.

References

[1] [n. d.]. Claude Code . https://claude.ai/.
[2] [n. d.]. Codex - AI assistant for work . https://chatgpt.com/codex/.
[3] [n. d.]. eBPF-based Networking, Observability, Security. https://cilium.io/.
[4] [n. d.]. Good programmers worry about data structures and their relationships. https://read.engineerscodex.com/p/good-programmers-worry-about-data.
[5] [n. d.]. Halting problem. https://en.wikipedia.org/wiki/Halting_problem.
[6] [n. d.]. Libbpf-tools benchmarks . https://github.com/iovisor/bcc/tree/master/libbpf-tools.
[7] [n. d.]. Meta Katran. https://github.com/facebookincubator/katran.

[8] [n. d.]. OpenEvolve - Evolutionary Coding Agent . https://github.com/algorithmicsuperintelligence/openevolve.

[9] Zhongjie Chen, Qingkai Meng, ChonLam Lao, Yifan Liu, Fengyuan Ren, Minlan Yu, and Yang Zhou. 2025. eTran: Extensible Kernel Transport with eBPF. In *USENIX NSDI*.

[10] Vishnu Asutosh Dasu, Monika Santra, Md Rafi Ur Rashid, Ashish Kumar, Saeid Tizpaz-Niari, and Gang Tan. 2026. Heimdall: Formally Verified Automated Migration of Legacy eBPF Programs to Rust. arXiv:2605.25411 [cs.CR] https://arxiv.org/abs/2605.25411

[11] Xiangyu Gao, Xiangfeng Zhu, Bhavana Vannarth Shobhana, Yiwei Yang, Arvind Krishnamurthy, and Ratul Mahajan. 2025. Offloading the Tedious Task of Writing eBPF Programs. In *Proceedings of the 3rd Workshop on EBPF and Kernel Extensions*.

[12] Robert Joseph George, Jennifer Cruden, Will Adkisson, Xiangru Zhong, Huan Zhang, and Anima Anandkumar. 2026. Torchlean: Formalizing neural networks in lean. *arXiv preprint arXiv:2602.22631* (2026).

[13] Yi He, Roland Guo, Yunlong Xing, Xijia Che, Kun Sun, Zhuotao Liu, Ke Xu, and Qi Li. 2023. Cross Container Attacks: The Bewildered eBPF on Clouds. In *USENIX Security*.

[14] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. 2018. The eXpress data path: fast programmable packet processing in the operating system kernel. In *ACM CoNEXT*.

[15] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with AlphaCode. *Science* (2022).

[16] Jinsong Mao, Hailun Ding, Juan Zhai, and Shiqing Ma. 2024. Merlin: Multi-tier Optimization of eBPF Code for Performance and Compactness. In *ACM ASPLOS*.

[17] Congcong Miao, Zhizhen Zhong, Yunming Xiao, Feng Yang, Senkuo Zhang, Yinan Jiang, Zizhuo Bai, Chaodong Lu, Jingyi Geng, Zekun He, Yachen Wang, Xianneng Zou, and Chuanchuan Yang. 2024. MegaTE: Extending WAN Traffic Engineering to Millions of Endpoints in Virtualized Cloud. In *ACM SIGCOMM*.

[18] Pedro Mizuno, Kimiya Mohammadtaheri, Linfan Qian, Joshua Johnson, Danny Akbarzadeh, Chris Neely, Mario Baldi, Nachiket Kapre, and Mina Tahmasbi Arashloo. 2025. A Target-Agnostic Protocol-Independent Interface for the Transport Layer. *arXiv preprint arXiv:2509.21550* (2025).

[19] Tian Pan, Enge Song, Yueshang Zuo, Shaokai Zhang, Yang Song, Jiangu Zhao, Wengang Hou, Jianyuan Lu, Xiaoqing Sun, Shize Zhang, Ye Yang, Jiao Zhang, Tao Huang, Biao Lyu, Xing Li, Rong Wen, Zhigang Zong, and Shunmin Zhu. 2025. Hermes: Enhancing Layer-7 Cloud Load Balancers with Userspace-Directed I/O Event Notification. In *ACM SIGCOMM*.

[20] Junxian Shen, Han Zhang, Yang Xiang, Xingang Shi, Xinrui Li, Yunxi Shen, Zijian Zhang, Yongxiang Wu, Xia Yin, Jilong Wang, Mingwei Xu, Yahui Li, Jiping Yin, Jianchang Song, Zhuofeng Li, and Runjie Nie. 2023. Network-Centric Distributed Tracing with DeepFlow: Troubleshooting Your Microservices in Zero Code. In *ACM SIGCOMM*.

[21] Qiongwen Xu, Michael D. Wong, Tanvi Wagle, Srinivas Narayana, and Anirudh Sivaraman. 2021. Synthesizing safe and efficient kernel extensions for packet processing. In *ACM SIGCOMM*.

[22] John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems* 37 (2024), 50528–50652.

[23] Yiwei Yang, Xiangyu Gao, Yuan Zhou, Yuhang Gan, Yusheng Zheng, and Andi Quinn. 2026. GPUOS: A GPU Operating System Primitive for Transparent Operation Fusion. *arXiv preprint arXiv:2604.17861* (2026).

[24] Yusheng Zheng, Yiwei Yang, Maolin Chen, and Andrew Quinn. 2024. Kgent: Kernel Extensions Large Language Model Agent. In *Proceedings of the ACM SIGCOMM 2024 Workshop on EBPF and Kernel Extensions*.

[25] Yusheng Zheng, Tong Yu, Yiwei Yang, Yanpeng Hu, Xiaozheng Lai, Dan Williams, and Andi Quinn. 2025. Extending Applications Safely and Efficiently. In *USENIX OSDI*.

[26] Yusheng Zheng, Tong Yu, Yiwei Yang, Minghui Jiang, Xiangyu Gao, Jianchang Su, Yanpeng Hu, Wenan Mao, Wei Zhang, Dan Williams, and Andi Quinn. 2025. gpu_ext: Extensible OS Policies for GPUs via eBPF. *arXiv preprint arXiv:2512.12615* (2025).

[27] Tal Zussman, Ioannis Zarkadas, Jeremy Carin, Andrew Cheng, Hubertus Franke, Jonas Pfefferle, and Asaf Cidon. 2025. cache_ext: Customizing the Page Cache with eBPF. In *ACM SOSP*.