Offloading the Tedious Task of Writing eBPF Programs

Paper # 74, 6 pages

ABSTRACT

eBPF offers a lightweight method to extend the Linux kernel without modifying source code in existing kernel modules. However, writing correct and efficient eBPF programs is hard due to strict constraints and cumbersome debugging processes. To tackle such an obstacle, we present SimpleBPF to offload the tedious eBPF development task. Developers only need to express their intent in a high-level domain-specific language without worrying about eBPF code generation details. SimpleBPF integrates four key components: a concise DSL, an LLM-based generator, a semantic checker, and an LLM-based optimizer. We use few-shot prompting and test SimpleBPF over programs written by one selected DSL. The result shows that SimpleBPF can successfully generate valid eBPF programs that pass the kernel verifier and exhibit competitive runtime performance. We also outline future directions based on current findings.

1 INTRODUCTION

Modern applications increasingly require customized kernel-level functionalities to meet the demand of high-performance networking [8, 13], security monitoring [9], and system observability [20, 21]. Due to strict security requirements from the operating system kernel, developers are typically not granted access to modify the kernel source code. The long development cycles and release timelines of upstream kernel maintainers make it impractical for users to wait for new features to be officially deployed/released. As a response, extended Berkeley Packet Filter(eBPF) [1] has emerged as a powerful mechanism for extending the operating system's kernel functionality.

Even though eBPF is a lightweight kernel extension approach, writing correct eBPF programs that are allowed to run within the kernel is not an easy task. Specifically, to ensure a safe and efficient execution, all eBPF programs need to pass a verifier. There are various strict constraints (e.g., avoid risky memory access, no unbounded loop) in the verifier, making eBPF programs writing a complex and error-prone task. Developers must understand the verifier's implicit rules and iteratively rewrite their eBPF programs to conform to the verifier's safety requirements, leading to a slow down of the development speed. To make thing more complex, different ways of writing an eBPF program can lead to varying JIT-compilation results and execution performance.

Currently, most of the efforts in eBPF ecosystem are centered around exploring new application scenarios where eBPF programs can play a role. Besides, works such as K2 [18] and Merlin [14] optimizes the generated bytecode for eBPF programs written in high-level source languages (like C or Rust). Comparatively little attention has been paid to reducing the manual efforts to write high-quality eBPF source programs. To the best of our knowledge, Kgent [19] is among the few existing works that leverage large language models (LLMs) for eBPF code generation from natural language, but it compromises on accuracy, limiting its practical usability. We believe that lowering the barrier to developing eBPF programs can help expand the usage scenarios of eBPF. Therefore, we propose to offload the task of writing eBPF programs to an automatic code generation system, SimpleBPF, which abstracts out all constraints and enables developers to only focus on correctly expressing their algorithm.

Building such a system faces multiple challenges. On the one hand, it should offer developers a more convenient way to write the code; On the other hand, it needs to guarantee the semantic equivalence and good execution performance. Accordingly, we design 4 main components in SimpleBPF to achieve this goal. First of all, SimpleBPF offers a domain-specific language (DSL) for developers to express their customized functionalities in a simpler way (e.g., fewer lines of code, predefined function libraries). Secondly, an LLM-based code generator is used to generate the eBPF expression that can pass the verifier. Thirdly, a Z3-based [10] semantic checker exists to ensure the semantic equivalence between the output eBPF program and the specification. Finally, an LLM-based optimizer further improve the eBPF program's execution performance by transforming it into a format that uses fewer instructions required by the JIT compiler. We use LLMs to leverage their strong generalization capabilities. Instead of writing rewrite rules, users can simply provide training examples to guide LLM to do code generation.

We choose the few-shot prompting and do a preliminary evaluation(§6) over SimpleBPF. The result shows that SimpleBPF can generate correct eBPF programs that pass the verifier from high-level DSL specifications that use about 55% fewer lines of code on average. SimpleBPF's optimizer further reduces the number of instructions for eBPF execution by 35% on average. We also outline several future directions (§7) such as high-level DSL design for more domains, effective feedback, and automatic hook selection.

2 PROBLEM STATEMENT

Writing “good" eBPF programs is hard because of several reasons below:

Programming eBPF requires engineers to learn unfamiliar coding patterns. Even if we can regard eBPF as a C-like language, it is not obvious for developers who are familiar with conventional languages to switch to eBPF. Common tasks—like accessing a hash map—require the use of special helper functions (e.g., `bpf_map_lookup_elem`) and explicit null checks, rather than simple indexing or dictionary-like syntax. As a result, developers must learn an entirely new programming discipline, which includes writing code in a verifier-compliant style that can seem unnatural.

The eBPF verifier may falsely reject valid programs due to overly strict constraints. To ensure the execution safety, the verifier imposes numerous constraints, such as prohibiting unbounded loops and requiring all memory accesses to be provably safe. Its conservative constraints force developers not only to ensure programs' functional correctness, but also to conform to a particular coding style and structural pattern. We argue that such an additional burden, optimizing programming style for verifier acceptability, introduces unnecessary overhead and hampers developer productivity.

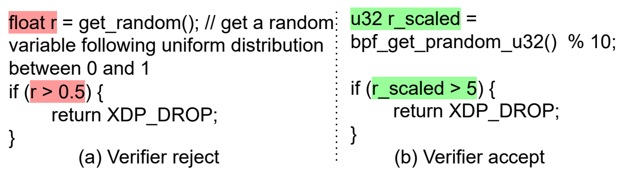
Execution performance is dependent on the written style. Unlike mature compilers that can optimize inefficient patterns, the JIT compiler performs minimal transformations for eBPF bytecode. As a result, two semantically equivalent programs can have various execution performance, depending on how the JIT backend interprets their structure. To achieve good execution performance (one of the most important reasons for people to use eBPF), developers need to carefully craft their programs to be JIT-friendly.

Debugging eBPF programs is slow. Given all these complexities, debugging eBPF to pass the verifier is quite common. Traditional software development environments (e.g., C and Python) offer mature debugging tools like GDB, pdb, and integrated step-by-step execution in IDEs (e.g., VSCode), but eBPF lacks such interactive tooling. Developers have to rely on runtime techniques such as `bpf_trace_printk()` or custom tracepoints to infer program behavior. Moreover, since the verifier rejects invalid programs before execution, developers frequently engage in a trial-and-error process to satisfy implicit constraints without clear guidance.

We list several snippets of bad and good eBPF programs to illustrate the mentioned difficulties mentioned above.

2.1 Necessary rewrite to pass the verifier

2.1.1 Type conversion. eBPF program does not support operations over types such as floating points. In fact, floating-point variables are commonly used in network functions such as load-balancing or rate limiting. For example, the fault injection network function compares a random value against a threshold to decide whether to drop a packet. Figure 1(a) shows an example of dropping a network packet with 50% probability rate, which cannot be expressed by eBPF without any type conversion. In order to represent this functionality, we need to use integer variables to replace floating-point variables. Figure 1(b) provides one option to replace floating-point variables by integer variables. Specifically, it scales up the variable's value by 10×, and the corresponding literal's value in the comparison condition is also multiplied by 10.



```
float r = get_random(); // get a random
variable following uniform distribution
between 0 and 1
if (r > 0.5) {
    return XDP_DROP;
}
```

(a) Verifier reject

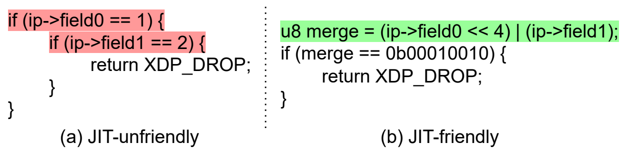
```
u32 r_scaled =
bpf_get_prandom_u32()  % 10;

if (r_scaled > 5) {
    return XDP_DROP;
}
```

(b) Verifier accept

Figure 1: Turn float operations to integer operations.

2.2 General optimization for faster verification and better execution

2.2.1 Combine ITE branches. The number of condition branches is an important factor to decide the complexity of an eBPF program. Each additional condition would exponentially increase the execution path from the entry to the exit of a program. A program with too many execution paths can degrade performance, increase verification complexity, and in the worst case, cause the verifier to reject it.



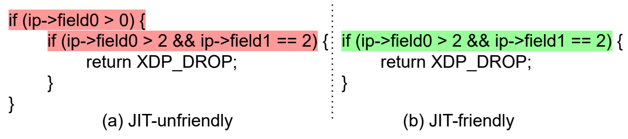
```
if (ip->field0 == 1) {
    if (ip->field1 == 2) {
        return XDP_DROP;
    }
}
```

(a) JIT-unfriendly

```
u8 merge = (ip->field0 << 4) | (ip->field1);
if (merge == 0b00010010) {
    return XDP_DROP;
}
```

(b) JIT-friendly

Figure 2: Branch reduction via condition merging.



```
if (ip->field0 > 0) {
    if (ip->field0 > 2 && ip->field1 == 2) {
        return XDP_DROP;
    }
}
```

(a) JIT-unfriendly

```
if (ip->field0 > 2 && ip->field1 == 2) {
    return XDP_DROP;
}
```

(b) JIT-friendly

Figure 3: Remove redundant ITE predicates.

Figure 2 and Figure 3 show 2 possible ways to merge multiple conditions. One approach is to combine all variables that are used in predicates into a temporary variable and then comparing the temporary variable against a constant. This offers the benefit of putting all conditions into one. Another strategy involves checking the logical relationships among predicates and removing redundant ones due to being supersets of others. They both improve the code execution performance without breaking the semantic equivalence.

Even though these optimizations are not that complex, the current lightweight JIT-compiler performs only basic bytecode-to-machine-code translation. Developers are responsible for manually restructuring their eBPF code in exchange for better performance.

2.2.2 Declare variables only when necessary. Register is one of the scarce resources in eBPF. Specifically, eBPF provides only 11 general-purpose 64-bit registers (r0 to r10) [2], and some of them are reserved. As a result, developers should be cautious for variable declaration because declaring unused or long-lived variables can increase the number of live registers at any point in the program. This would unnecessarily increase the state space that the verifier has to track.

```
int a = 0; int b = 0;
if (cond) { a = compute_a();
} else { b = compute_b();
}
return a + b;
```

(a) JIT-unfriendly

```
if (cond) {
    int a = 0; a = compute_a();
    return a;
}
int b = 0; b = compute_b();
return b;
```

(b) JIT-friendly

Figure 4: Do variable declaration when needed.

A good eBPF program should declare the variable only when needed. For instance, as is shown in Figure 4, instead of declaring `variable a and b` in the beginning, we should declare them within the branch because some declarations can be avoided if certain conditions are not satisfied.

2.3 Domain-specific optimizations

2.3.1 Take into consideration the workload feature. These optimizations exist only when developers know beforehand some features in a specific domain. Concretely, in eBPF, short-circuit evaluation of logical && is preserved in the generated bytecode. Therefore, if domain knowledge suggests that one condition is more likely to fail, placing it before others can reduce the number of instructions executed at runtime.

For instance, if the developers know that `ip->field0` is more likely to be less than or equal to 2 in Figure 5, checking this condition earlier is preferable for execution.

```
if (ip->field1 == 2 && ip->field0 > 2) {
    return XDP_DROP;
}
```

(a) runtime suboptimal

```
if (ip->field0 > 2 && ip->field1 == 2) {
    return XDP_DROP;
}
```

(b) runtime optimal

Figure 5: When we know from the domain knowledge that `ip->field0 > 2` is more likely to be false, it is better to check this condition first.

3 ABSTRACTING WITH TAILORED DSLS

Given the difficulties of writing verifier-friendly and JIT-optimal eBPF programs, we advocate tailoring the domain-specific languages (DSLs) design paired with a code generator that automatically translates high-level intent into an efficient eBPF expression. Writing in a DSL allows developers to focus on high-level intent without dealing with the low-level constraints of the eBPF verifier or specific optimizations. Besides, we have the freedom to make the DSL closely resemble the language commonly used by domain experts, minimizing their learning curve.

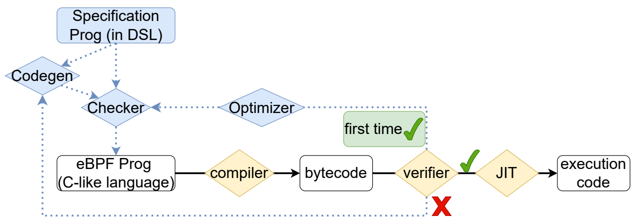


Figure 6: Our proposal: developing a new DSL and an LLM-based code generator to output the eBPF program. After passing the semantic checker and verifier, an LLM-based optimizer is used to continue optimizing the program to improve execution performance. Blue components are related to our proposal while yellow components exist for eBPF compilation.

Figure 6 shows a proposed workflow of SimpleBPF. Developers express their customized DSL programs. Afterwards, these programs are fed into an LLM-based code generator and a semantic checker to output eBPF programs that have the same semantics. If the output program passes the verifier, an LLM-based optimizer continues to optimize the eBPF program to improve its execution performance. Otherwise, the code generator outputs another candidate for the verifier.

SimpleBPF separates the code generator and optimizer into 2 parts instead of combining them because of several benefits. *(1) Each LLM prompt can be tailored specifically to either generation or optimization, allowing the model to better attend to the relevant patterns. (2) This avoids unnecessary work such as optimizing semantically incorrect code output from the generator.* SimpleBPF is retargetable by supporting various DSLs as long as developers provide sufficient training data and a corresponding semantic equivalence checker.

We also consider designing a general-purpose language (GPL) for eBPF code generation, but doing so offers limited practical advantage. eBPF itself resembles a constrained general-purpose environment, so building another GPL layer on top does not offer too many abstraction benefits. Moreover, GPLs are inherently harder to learn, as learners must grasp a wide range of constructs that may be irrelevant to their specific use cases. In contrast, domain experts already possess deep knowledge of the particular problem space (e.g., rpc requests in microservice, lookup query in database), and hence can adopt the corresponding DSL with fewer efforts.

4 RESEARCH QUESTIONS

To generate valid and performant eBPF programs automatically, we need to address a few following research questions.

Q1: *What are key features that a high-level language should provide?* The goal of designing a new high-level language is to provide eBPF developers with a more convenient tool instead of worrying about the constraints (e.g., no unbounded loop). To realize this goal, we should take into consideration several aspects. First of all, *expressiveness*. The designed language should cover operations that are allowed by existing eBPF programs. Second, *simplicity*. It offers an easier way to express the same functionality. This simplicity can be measured by the lines of code (loc). Third, *flexibility*. Ideally, we want the language design to be flexible enough to enable developers to provide hints (e.g., eBPF data structure choice) to guide the code generator for better program output.

Q2: *How to build the code generator?* To evaluate different ways to develop the code generator, we need to consider several metrics. First, *efficiency*. It measures the speed for this generator to produce code in deployment. Second, *correctness*. Whether the generated code preserves the semantic of the specification. Third, *development effort*. This refers to the difficulty to implement and maintain the code generator. These metrics serve as guiding principles across approaches.

Q3: *How to ensure the correctness of the code generator's output?* We propose building a semantic checker that systematically verifies whether the generated code faithfully implements the specification. This is a challenging task because it requires formalizing the semantics of both the source DSL and the target eBPF program. These models should take into consideration the algorithm functionality, low-level memory access, and nondeterministic behavior (e.g., random value generation). Whether through symbolic execution, SMT-based equivalence checking, or test-based differential analysis, this checker is an indispensable part of SimpleBPF.

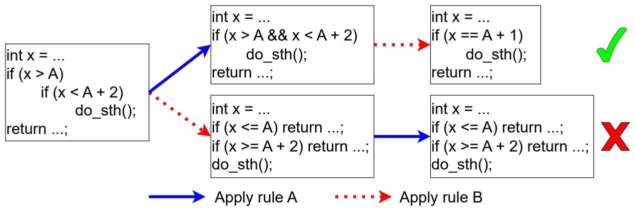
Q4: *How to integrate with existing eBPF ecosystem?* The existing eBPF ecosystem is widely adopted, supported by a large and active community of developers. Instead of reinventing the ecosystem from scratch, our goal is to complement and extend the existing development workflows. For example, developers should have the flexibility to either program directly in C-like eBPF syntax or start writing the newly developed DSL. Regardless of which path they choose to write programs, developers benefit from SimpleBPF.

5 A POTENTIAL APPROACH

In this section, we want to partially answer questions in §4 through a description of SimpleBPF design (shown in Figure 6). It consists of 4 main parts: a DSL, an LLM-based code generator, a Z3-based checker, and an LLM-based optimizer. DSL offers developers a convenient way to express their algorithms; the code generator and semantic checker work together to ensure the semantic equivalence; finally, the optimizer optimizes the eBPF code into a JIT-friendly format that is more performant to execute.

5.1 DSL and code generator

To make eBPF programming more accessible to domain experts, we choose to design a domain-specific language (DSL) that is simple, expressive, and closely aligned with existing domain terminology. We also have the flexibility to predefine commonly used functions, allowing developers to focus on high-level logic rather than implementing low-level operations from scratch. Additionally, the DSL design should be extensible, providing room for incorporating new features and abstractions as domain requirements evolve.



```
int x = ...
if (x > A)
      if (x < A + 2)
            do_sth();
return ...;
```

```
int x = ...
if (x > A && x < A + 2)
      do_sth();
return ...;
```

```
int x = ...
if (x == A + 1)
      do_sth();
return ...;
```

```
int x = ...
if (x <= A) return ...;
if (x >= A + 2) return ...;
do_sth();
```

```
int x = ...
if (x <= A) return ...;
if (x >= A + 2) return ...;
do_sth();
```

Figure 7: Rule A → is condition merging within an ITE while Rule B ⇢ is early return. For the given example, applying Rule B first prevents us from reaching the optimal result that is possible if Rule A were applied before Rule B.

To translate DSL programs into verifier-compliant and JIT-friendly eBPF code, we consider using the LLM-based code generation approach by leveraging state-of-the-art LLM APIs [15] [11] [12]. The current development of language models brings LLM-based code generation several unique advantages. Compared to synthesis-based techniques, LLMs offer faster inference and broader applicability as not all translation tasks can be encoded into tractable synthesis problems. Compared to the rigid rule-based program rewriting, LLM-based solution can offer more flexibility to explore outcomes with fewer manual efforts involved. Concretely, there are 2 rewrite rules (A and B) in Figure 7, and their application order can affect the output code quality. In general, it is not surprising that a rule-based code generator might output suboptimal results [17], often due to an incomplete set of rewrite rules or a suboptimal application order. Moreover, humans are slow at encoding rewrite rules into code, whereas LLMs can rapidly generalize from some human-crafted examples to automate similar transformations for new examples. LLMs might introduce extra challenges, such as accuracy loss and high training cost. Addressing these issues is essential for the system's development.

5.2 Semantic checker

A semantic checker is necessary to ensure the correctness of the generated eBPF programs. This checker takes as an input the DSL program and the generated eBPF program and checks whether they express the same functionality. There are multiple ways to compare semantics between different programs. A heavy-weight method applies formal proof tools like Coq to offer strong guarantees, but this requires significant manual proof effort and expertise. A light-weight approach tests selected input/output pairs. This is easier to implement but may miss subtle semantic discrepancies.

```
def program_A(Input) --> return Output_A
def program_B(Input) --> return Output_B
s = Solver()
s.add(program_A(Input) != program_B(Input))
if s.check() == sat --> NOT equivalent.
else: --> equivalent.
```

Figure 8: Semantic checker for equivalence.

These 2 main methods are not mutually exclusive. There could be a hybrid approach: employing formal methods for critical parts of the program, while relying on input-output test cases to cover other parts. Here, we choose to turn input and output program into SMT formula for equivalence checking. As is shown in Figure 8, we use `program_A` and `program_B` to represent input and output programs' functionality and use Z3 to search for counterexamples that demonstrate behavioral differences. If there are no such counterexamples, we conclude that they are equivalent.

5.3 The eBPF optimizer

We propose building an LLM-based optimizer. We also consider other alternatives but rule-based optimizers require extensive manual effort to design and maintain. Synthesis-based optimizers [18] can be computationally expensive and slow to scale. Our method mirrors the structure of our LLM-based code generator: instead of DSL-to-eBPF examples, we now train the model on pre-optimization and post-optimization eBPF program pairs (e.g., examples in §2.2), allowing the model to learn optimization patterns directly from data. This approach offers a more scalable and automation-friendly way to produce optimized eBPF code that improves the execution performance. For those who choose to write an eBPF program directly, this optimizer is a valuable addition by turning their written program into a format that achieves better execution performance.

6 CASE STUDY

6.1 Experiment setup

We choose an existing DSL, AppNet [22], specifically designed for service mesh functions, as the basis for our evaluation. An AppNet example is shown in Figure 9 to express the behavior of randomly dropping an RPC request. A typical AppNet program consists of multiple parts: `state` lists all global variables, `init()` initializes all global variables, and `req(rpc)` presents the service mesh function when receiving one RPC request. Detailed language design is explained in AppNet paper [22].

```
state:
   prob: float
init():
   prob = 0.95
req(rpc):
   match randomf(0, 1) < prob:
      true => send(rpc, Down)
      false => send(err('fault_injected'), Up)
```

Figure 9: An AppNet example that drops RPC requests with 95% probability.

We adopt in-context learning of prompt engineering to build the LLM-based code generator and optimizer. We choose this approach at this moment instead of other alternatives (e.g., fine-tuning) because in-context learning is a good option to start exploring a new problem space [4]. After constructing training examples consisting of AppNet–eBPF program pairs, as well as pre-optimized and post-optimized eBPF program pairs, we pass these training data through the ChatGPT 4o interface to guide its learning (e.g., floating point $\longrightarrow$ integer, early exit). We evaluate the synthesized eBPF code and assess the effectiveness of optimizations.

6.2 Preliminary results.

Benchmarks and baseline. We generate eBPF implementation for AppNet programs that describe the application network functions to deal with RPC requests in microservices. Previous AppNet compiler targets 3 RPC processing platforms: gRPC interceptors [5] and EnvoyNative [6], Envoy-Wasm [7]. We write 3 AppNet programs in Table 1 for testing. To be specific, logging means the AppNet program maintains some global variables and updates their value when receiving one RPC request; fault injection conditionally drops requests based on specified criteria. To increase complexity, we construct these programs using multiple global variables and compound conditional expressions. Functionalities of all benchmarks [3] are independent of the RPC payload. The task of deserializing payloads from gRPC packets is orthogonal to the contributions of this paper.

Table 1: Evaluate different components of SimpleBPF over benchmarks [3]. Green represents better results.

AppNet Program		Rule-based code generator			LLM-based code generator			Post optimization eBPF		
Name	**Loc**	**Loc**	**# eBPF instr.**	**# JIT instr.**	**Loc**	**# eBPF instr.**	**# JIT instr.**	**Loc**	**# eBPF instr.**	**# JIT instr.**
Logging (2 vars)	11	32	125	141	24	67	67	22	67	67
Logging + Fault injection	12	36	119	145	36	119	145	29	79	101
Fault injection (Optimizable condition)	9	26	59	76	26	59	76	20	43	56

We built a rule-based code generator by extending the AppNet compiler to target the eBPF backend. This incorporates basic program rewrite rules but does not apply any code optimizing algorithms. We compare it against SimpleBPF and quantify the benefits brought by the LLM-based optimizer.

Assumptions. In this work, we target the XDP hook on the sender side for all generated eBPF programs, as the selected AppNet programs are designed for sender-side network functions. For simplicity, we assume each RPC request fits within a single network packet to bridge the semantic gap between AppNet, which operates at the granularity of RPC requests, and eBPF, which processes data at the granularity of individual packets. Generating eBPF programs for RPC requests that span across multiple packets is left for future work.

Preliminary results. We measure the performance of 2 code generator and LLM-based optimizer over 3 main metrics: loc, # eBPF instructions for eBPF bytecode, and # JIT instructions for execution. Loc measures the easiness to write the program while others determine the actual performance of eBPF program. Rule-based code generator only applies several rewrite rules without implementing any optimizations. We provide basic examples (with no optimization hints) for LLM-based code generator to learn rewrite patterns.

According to the results in Table 1, AppNet *reduces the loc by 45.5% on average*; the quality of post-optimization eBPF programs is better, meaning that both LLM-based code generator and LLM-based optimizer contributes to eBPF code optimizing, *by reducing # eBPF instructions and JIT instructions by around 64% on average.* We want to share 2 extra findings. (1) writing an LLM-based code generator significantly reduces development effort, as crafting a few-shot prompt for in-context learning requires far fewer loc than implementing a full rule-based system (several hundreds loc vs several thousands loc). (2) even though LLM-based optimizer further optimizes the output code from LLM-based code generator, such improvement may not continue reduce the instructions since some of the optimization rules are already applied during LLVM bytecode generation. We confirm that generated eBPF programs can pass the verifier and verify their semantic equivalence in Z3.

7 FUTURE WORK

We list directions for improvements beyond SimpleBPF.

Optimal hook selection. Hook selection [16] is essential in eBPF code generation because different hooks lead to different performance (e.g., latency). SimpleBPF assumes that eBPF hooks are preselected. In the future, we want to continue offloading developers' burden by automatically choosing the suitable hook that gives the best performance.

DSL design for other domains. The case study focuses on generating eBPF programs for service mesh functions. SimpleBPF can be extended to new domains (e.g., database query and system monitoring). DSL for these domains can bring unique challenges and opportunities for building domain-specific code generation and optimization models.

Effective feedback from semantic checker and verifier. We use a checker to verify the semantic correctness of eBPF code. When mismatches are detected, SimpleBPF simply restarts the generation process, which misses the opportunity to provide targeted and constructive feedback to guide the model. An intriguing direction is to design effective mechanisms for generating meaningful feedback signals from the checker, enabling the LLM to refine its code generation and optimization decisions in an interactive manner.

8 RELATED WORKS

Recent efforts on generating and optimizing eBPF programs fall into 2 main categories: natural-language–driven source code generation and bytecode-level optimization.

Kgent [19] leverages large language models to translate informal, English descriptions into eBPF, dramatically reducing the manual development effort. However, Kgent does not provide checker to assure the semantic equivalence. K2 [18] and Merlin [14], by contrast, focus on optimizing eBPF bytecode by introducing an additional optimization pass that produces semantically equivalent but more performant versions for the verifier. Our work is complementary by addressing the problem of generating eBPF programs from high-level DSLs.

9 CONCLUSION

We propose a new system, SimpleBPF, that contains a high-level DSL and an affiliated LLM-based code generator to offer a more convenient way for eBPF development. Preliminary results show that people can write simpler code in DSL, and programs generated by few-shot prompt engineering outperform the rule-based model in terms of # instructions required for execution. We hope our proposal can encourage more research on easier eBPF development.

REFERENCES

[1] eBPF. https://ebpf.io/. (Accessed on 04/13/2025).

[2] eBPF Instruction Set Specification, v1.0. https://www.ietf.org/archive/id/draft-thaler-bpf-isa-00.html.

[3] Eval benchmarks. https://anonymous.4open.science/r/SimpleBPF_benchmark-4B59/README.md.

[4] Fine-Tuning vs. In-Context Learning: A Practical Guide. https://medium.com/@heyamit10/fine-tuning-vs-in-context-learning-a-practical-guide-08163ede6d1a.

[5] grpc interceptors. https://grpc.io/docs/guides/interceptors/.

[6] Http filters. https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/http_filters.

[7] Webassembly in envoy. https://github.com/proxy-wasm/spec/blob/main/docs/WebAssembly-in-Envoy.md.

[8] IOVisor Authors. Introduction to express data path. https://www.iovisor.org/technology/xdp. Accessed: 2025/05/05.

[9] Linux Authors. Introduction to ebpf lsm. https://docs.kernel.org/bpf/prog_lsm.html. Accessed: 2025/05/05.

[10] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, 2008.

[11] Google DeepMind. Gemini 1 model card. https://deepmind.google/technologies/gemini/, 2023. Accessed: 2025-05-18.

[12] DeepSeek. Deepseek-coder: A family of open-source code language models. https://github.com/deepseek-ai/DeepSeek-Coder, 2023. Accessed: 2025-05-18.

[13] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. {BMC}: Accelerating memcached using safe in-kernel caching and pre-stack processing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 487–501, 2021.

[14] Jinsong Mao, Hailun Ding, Juan Zhai, and Shiqing Ma. Merlin: Multi-tier optimization of ebpf code for performance and compactness. In *ACM ASPLOS*, 2024.

[15] OpenAI. Gpt-4 technical report. https://arxiv.org/abs/2303.08774, 2023. Accessed: 2025-05-18.

[16] Farbod Shahinfar, Sebastiano Miano, Giuseppe Siracusano, Roberto Bifulco, Aurojit Panda, and Gianni Antichi. Automatic kernel offload using bpf. In *ACM HotOS*, 2023.

[17] Eelco Visser. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*, 2005.

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

[19] Yusheng Zheng, Yiwei Yang, Maolin Chen, and Andrew Quinn. Kgent: Kernel extensions large language model agent. In *Proceedings of the ACM SIGCOMM 2024 Workshop on EBPF and Kernel Extensions*, 2024.

[20] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, et al. {XRP}:{In-Kernel} storage functions with {eBPF}. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 375–393, 2022.

[21] Yang Zhou, Xingyu Xiang, Matthew Kiley, Sowmya Dharanipragada, and Minlan Yu. {DINT}: Fast {In-Kernel} distributed transactions with {eBPF}. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 401–417, 2024.

[22] Xiangfeng Zhu, Yang Zhou, Yuyao Wang, Xiangyu Gao, Arvind Krishnamurthy, Sam Kumar, Ratul Mahajan, and Danyang Zhuo. High-level programming for application networks. In *USENIX NSDI*, 2025.