Kernel Extension DSLs Should Be Verifier-Safe!

Franco Solleza Justus Adam Akshay Narayan Malte Schwarzkopf
Andrew Crotty † Nesime Tatbul ‡

Brown University † *Northwestern University* ‡ *MIT & Intel*

Abstract

eBPF allows developers to write safe operating system extensions, but writing these extensions remains challenging because it requires detailed knowledge of both the extension's domain and eBPF's programming interface. Most importantly, the extension must pass the eBPF verifier.

This paper argues that DSLs for extensions should guarantee *verifier-safety*: valid DSL programs should result in eBPF code that always passes the verifier. This avoids complex debugging and the need for extension developers to be eBPF experts. We show that three existing DSLs for different domains are compatible with verifier-safety. Beyond verifier-safety, practical extension DSLs must also achieve good performance. Inspired by database query optimization, we sketch an approach to creating DSL-specific optimizers capable of maintaining verifier-safety. A preliminary evaluation shows that optimizing verifier-safe extension performance is feasible.

ACM Reference Format:
Franco Solleza, Justus Adam, Akshay Narayan, Malte Schwarzkopf, Andrew Crotty, Nesime Tatbul. 2025. Kernel Extension DSLs Should Be Verifier-Safe!. In *3rd Workshop on eBPF and Kernel Extensions (eBPF '25), September 8–11, 2025, Coimbra, Portugal.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3748355.3748368

1 Introduction

Kernel extensions give developers access into the operating system (OS) to perform customized kernel tasks like process scheduling [3], network congestion control [13], and detailed observability [18]. Kernel extension APIs like Linux kernel modules or eBPF facilitate development of such extensions. However, writing kernel extensions remains difficult today because developers face three challenges:

eBPF '25, September 8–11, 2025, Coimbra, Portugal

ACM ISBN 979-8-4007-2084-0/25/09
https://doi.org/10.1145/3748355.3748368

(1) Bespoke programming environment. The OS kernel is a unique programming environment with its own set of data structures, invariants, and APIs. Writing usable kernel extensions requires expertise programming in this environment.

(2) Unstable Kernel API. Since they extend the kernel's functionality, extensions rely on internal kernel interfaces and must keep up with frequent changes to the kernel APIs. Maintaining extensions becomes increasingly difficult, representing a frequent source of bugs and developer effort [5, 36].

(3) Safety. Kernel extensions execute in the kernel's execution context, so buggy extensions could compromise the entire kernel, including leaking data, accessing invalid memory, corrupting state, and degrading performance for the entire machine.

Today, extension developers often use domain-specific languages (DSLs) to write kernel extensions. DSLs simplify extension development because they obviate the need for developers to understand the details of the kernel's programming environment. DSLs also centralize the burden of keeping up with an unstable kernel API to DSL maintainers, so extension developers can write extensions that are portable across supported kernel versions. For example, the DSLs in BPFTrace [18], CCP [24], and Syrup [13] greatly simplify programming extensions for observability, congestion-control, and scheduling, respectively.

However, DSLs are not a panacea. While many DSLs for kernel extensions solve the first two challenges, they remain unsafe because they rely on carefully engineering the DSL's toolchain (*e.g.,* a kernel module) so that their extensions will not crash the OS. For example, bugs in the LTTng [6, 15] toolchain or CCP kernel module [24] can cause kernel crashes.

eBPF provides a restrictive environment and in-kernel verifier that statically verifies that an extension is safe to run. Thanks to this safety, extension developers increasingly use eBPF to extend the Linux kernel [3, 4, 9, 21–23, 25, 28, 31, 34, 35, 37]. eBPF is a promising path forward for kernel extension DSLs: unlike custom kernel modules, eBPF ensures that DSL toolchains and the eBPF programs they generate cannot cause OS bugs. However, this safety feature shifts most of the onus to extension developers, who must now reason about why an eBPF extension fails the verifier—a notoriously difficult task (§2), especially for non-experts. While some existing eBPF-based extensions expose DSLs [13, 18, 35], their compilers can generate eBPF programs that the verifier rejects. This means that despite using a DSL, the extension developer needs to

reason about eBPF's verifier. Thus, extension developers must spend time learning how to use eBPF effectively rather than implementing extension functionality.

A Call for Research on Verifier-Safe DSLs. DSLs should ideally provide *verifier-safe* programming environments; *i.e.,* the DSL compiler should only generate eBPF programs that the verifier will accept. With a verifier-safe DSL, non-experts can write kernel extensions without the risk of crashing the kernel or reasoning about complex eBPF verifier errors. §3 describes three strategies for addressing this challenge in the traditional eBPF setting, ranging from simply attempting various generated candidate programs to formally modeling the verifier. These strategies share drawbacks that stem from the design choice to operate over eBPF bytecode directly. Instead, we propose a new strategy, *DSL-guided iteration* (§3.2), that leverages common DSL properties to produce verifier-safe eBPF programs. We investigate whether practical DSLs have the properties necessary for DSL-guided iteration by considering kernel extension DSLs across the domains of observability, congestion control, and task scheduling (§4).

Verifier safety is distinct from performance, meaning that verifier-safe programs might perform poorly. Of course, performance is important, and we argue for verifier-safe optimization techniques for kernel extension DSLs (§5). Our preliminary evaluation (§6) of an observability DSL toolchain shows that verifier-safe DSLs can support performance comparable to that of extensions hand-optimized by eBPF experts.

2 Background

DSLs simplify kernel extension development because they decouple domain-specific functionality from kernel-specific APIs and provide a stable programming and data model that abstract kernel objects and APIs. While the DSL's creators must address these challenges when implementing the DSL, they only do so once. For example, CCP [24] supports portability for congestion control algorithms across multiple datapaths, and Floem [27] exposes features of NIC drivers using simple constructs. When executing the program in the kernel, the DSL determines the appropriate mapping between the data model and the target kernel's objects. For example, Osquery [26], DBOS [29], and OSDB [30] all propose using database principles (with SQL as the DSL) to manage OS state, each with a different data model that maps kernel objects to a database schema. Similarly, LTTng's toolchain [6] provides abstractions for extracting and analyzing Linux kernel traces.

Kernel extensions increasingly rely on eBPF to ensure that they avoid crashing the kernel [8], but this presents a new challenge: extension developers must write extensions (using the DSL) whose eBPF code passes the in-kernel verifier. Reasoning about why an eBPF extension fails the verifier is challenging [12, 36] for two reasons. First, the verifier's rules disallow common programming patterns (*e.g.,* large loops, deep nested function calls) and limit the extension's stack. Second, the verifier performs its checks on eBPF bytecode rather than a high-level programming language (*i.e.,* the DSL), so it is difficult to identify the failure's cause in the original program.

Consequently, writing extensions in current DSLs requires deep eBPF expertise. For example, BPFTrace [18] and Ply [20] are DSLs for observability that use eBPF, but their users can write code that fails the verifier. Meanwhile, eTran [2], PageFlex [35], and Syrup [13] expose eBPF's complexity directly to developers, so extensions in those systems can also fail the verifier. When a program in these DSLs fails the verifier, the developer must try to understand why it failed and how to change the DSL program so that it passes.

3 Achieving Verifier-Safety

This paper argues that eBPF-based DSLs for kernel extensions should be *verifier-safe*. In other words, they should guarantee that **any valid program written in the DSL will pass the verifier and run**. Figure 1 illustrates four techniques a DSL compiler could use to ensure that its generated eBPF programs are verifier-safe.[1] The first three, verifier iteration (Figure 1a), verifier modeling (Figure 1b), and verifier extraction (Figure 1c), operate over eBPF bytecode. We propose the fourth, DSL-guided iteration (Figure 1d). A DSL compiler could use any combination of these strategies in concert.

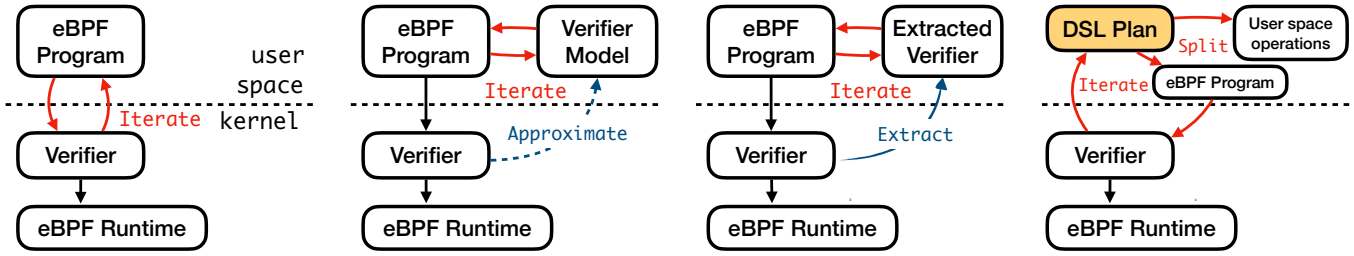
3.1 Bytecode-Oriented Strategies

Verifier Iteration. With verifier iteration, a DSL toolchain would generate several candidate programs and submit each candidate to the verifier until finding one that passes. The space of possible programs is large, and repeatedly querying the kernel verifier with arbitrary eBPF programs is prohibitively slow. However, while no practical system uses *only* verifier iteration, the Linux kernel's documentation states that "the only way to know that the program is going to be accepted by the verifier is to try to load it" [14], which means that any automated approach must ultimately use trial-and-error.

The fundamental problem with this strategy is that DSL compilers treat the verifier as an external black box and do not reason about whether their output bytecode is verifier-safe. The following strategies attempt to alleviate this limitation.

Verifier Modeling. Verifier modeling guides the toolchain to generate candidates that are likely to pass the verifier. In this approach, the toolchain maintains an approximate model of the constraints imposed by the kernel's verifier. With the model, the toolchain can evaluate whether candidate implementations are likely to pass the verifier and try only those that do while also optimizing for a specific target (*e.g.,* higher performance). For example, K2 [34] generates optimized versions of eBPF programs by modelling the verifier's constraints

[1] Some past work modifies the verifier [7] or other parts of the in-kernel eBPF toolchain [16], but this paper assumes an unmodified kernel.



a: **Verifier Iteration** b: **Verifier Modeling** c: **Verifier Extraction** d: **DSL-guided Iteration**

Figure 1: **Four strategies to verifier-safety. Strategies 1a–1c must reason about arbitrary programs, and so must consider large search spaces. DSLs impose structure on the search space. Strategy 1d thus combines aspects of both iteration and modeling.**

via SMT, which is used to determine whether the optimized program will pass. Unfortunately, given the lack of a verifier constraint specification, these models are inherently approximations. Even candidate programs the model says should pass can fail the actual verifier, leaving the developer with the challenge of fixing the program.

Verifier Extraction. As an improvement on verifier modeling, verifier extraction builds its model of the kernel verifier by mechanically transforming the verifier's source code. The toolchain will always produce verifier-safe code as long as the transformation method is complete. Prior works have used this approach successfully to prove the correctness of individual verifier components (*e.g.*, range analysis in Agni [33] using SMT formulas). Unfortunately, the size and complexity of the verifier makes the SMT formulas prohibitively expensive for online reasoning about a program's verifier-safety.

3.2 Our Proposal: DSL-Guided Iteration

The previous strategies all separate eBPF's verification step from compiling extension code. This fundamentally limits such strategies because they lose information about the high-level extension program. Instead, we propose a new strategy, DSL-guided iteration, that builds on two key insights.

Kernel-user space split. Since eBPF supports safely sending and receiving data between eBPF and user space, a DSL's toolchain can split extension functionality between these two domains. The user space component lacks access to kernel state and must rely on the eBPF component for input data, but it is also free from the verifier's restrictions. This makes it more flexible (*e.g.*, loops in user space are not restricted).

High level execution plan. The DSL is a higher-level representation than an eBPF program, which gives the toolchain freedom to determine how to translate a DSL program into eBPF code. The toolchain first represents the program with an execution plan—a sequence of high-level operations—and then chooses a concrete execution that splits the plan between user space and kernel space (*e.g.*, eBPF) programs.

Not all execution plan splits generate verifier-safe eBPF components. To ensure verifier-safety, execution plans must support a minimal split where the in-kernel components are represented by *minimum verifier-safe functions* (MVSFs). This can be achieved by creating MVSFs for the primitive DSL operations that must execute in the kernel. For example, for a kernel-tracing DSL those primitives read kernel state. An MVSF for such a primitive performs the read, then immediately communicates the data to the extension's user space component. Since the DSL has a finite set of primitive operations, there is a finite set of MVSFs, allowing DSL developers to check all MVSFs against the verifier ahead of time. With MVSFs, the DSL is implicitly verifier-safe—it can always generate an eBPF component that contains only the MVSF[2] and rely on the extension's user space component for any subsequent operation.

4 Verifier-Safety in DSLs

A kernel extension DSL that defines MVSFs is guaranteed to be verifier-safe. We show that defining MVSFs is feasible using DSLs in three domains as examples: congestion control, task scheduling, and observability. These DSLs already abstract kernel functionality and offer portability, so adding verifier-safety makes them meet the three criteria from §1.

4.1 Congestion Control

Congestion control plane (CCP) [24] provides a framework to implement congestion control algorithms outside the network datapath. Congestion control algorithm implementations in CCP use a DSL to express "datapath programs," which are parts of the algorithm's logic that must run in the datapath context because they access flow state (Figure 2a). CCP implements a compiler, bytecode format, and an interpreter in a kernel module to execute datapath programs. CCP programs cannot express loops or recursion, have a size limit, and can only access a predefined, compiler-checked set of measurement signals (*e.g.*, a flow's RTT). While using a bespoke bytecode format and interpreter eases portability across datapaths, replacing the kernel module with eBPF would improve performance and guarantee kernel safety. Indeed, Linux cites CCP as an inspiration for including congestion control algorithms in eBPF's functionality [17].

[2]Some DSLs (*e.g.*, the one in §4.3) require loading multiple MVSFs simultaneously. Since they are separate programs, their verifier-safety is unaffected.

```
(when (== pulseState 0)
  (:= Rate UpRate) (:= pulseState 1))
(when (&& (== pulseState 1)
          (> Timer.micros
             Flow.rtt_sample_us))
  (:= Rate DownRate)
  (:= pulseState 2))
```

a: **An excerpt of CCP's implementation of BBR's pulsing behavior.**

```
uint32_t schedule (
  void *pkt_start,
  void *pkt_end ) {
    uint32_t hash = hash(
      (struct *udphdr) pkt_start);
    return hash % NUM_EXECUTORS;
}
```

b: **An excerpt of Syrup's round robin scheduler using a subset of C.**

```
SELECT A.pid,
  A.syscall_number,
  MAX(A.exit_ts - A.enter_ts)
    AS latency
FROM system_calls AS A
GROUP BY A.pid, A.syscall_number
WINDOW 1sec
```

c: **A SQL query for finding high-latency syscalls.**

Figure 2: **Three example DSLs for kernel extensibility.**

Verifier-Safety. An MVSF in an eBPF-based CCP program might simply query and report congestion-related measurements; in fact, many CCP-based algorithms (*e.g.,* Nimbus [10]) already use such minimal programs.

DSL-based iteration also enables extensions to CCP's functionality while maintaining verifier-safety. For example, CCP users currently need to hard-code a congestion control algorithm's execution plan. Instead, the CCP toolchain could generate an execution plan, which would simplify algorithm implementation. Not all components of CCP algorithm implementations can run in eBPF, since CCP supports algorithms with asynchronous or unsupported operations (*e.g.,* floating-point). So, an eBPF-based implementation of CCP would need to decide how to split work between user space and eBPF. For complex algorithms like BBR [1], the toolchain could then push more algorithm functionality into the datapath program.

4.2 Task Scheduling

Syrup [13] provides a framework for implementing application-specific scheduling policies. It models task scheduling as a matching problem between events to process and executors to process them. Syrup users write matching functions in a subset of C (Figure 2b) and insert them at various levels of the stack (*e.g.,* in a SmartNIC to implement packet scheduling). Syrup can run scheduling policies as eBPF programs in the kernel or make complex scheduling decisions entirely in user space by leveraging ghOSt [11]. Developers choose between these options based on performance and functionality requirements: eBPF-based scheduling programs are more efficient, but user space ones are more flexible. When targeting eBPF, developers currently must manually ensure that their scheduling program passes the verifier.

Verifier-Safety. Syrup already offers a data model, based on maps, that task scheduling programs use. However, Syrup's DSL requires modification to make it verifier-safe. In particular, Syrup's DSL allows for arbitrary pointer operations, which lets the DSL express fundamentally unsafe programs that will not pass the verifier. Fortunately, removing pointer operations from Syrup's DSL is feasible without compromising functionality. For example, Figure 2b might be rewritten in such a (future) DSL as follows:

```
schedule(Packet):
  hash(Packet.udp_header) % NUM_EXECUTORS
```

Current Syrup supports two extremes: *(i)* use MVSFs to read scheduler events and pass information to user space, where scheduling code runs, or *(ii)* put all scheduler decision making in the kernel. An improved version of Syrup based on DSL-guided iteration could make the boundary more fluid and automatically push parts of the scheduling program into eBPF.

4.3 Observability

SQL-like DSLs are commonly used for observability queries (*i.e.,* tracing how OS code affects an application's behavior). In these approaches to observability (*e.g.,* OSQuery [26]), an observability engineer writes a query against an abstract, well-defined schema akin to a database schema. A toolchain supporting SQL-like queries (Figure 2c) can translate its data model—*i.e.,* a `system_calls` table, with columns such as `pid` (process ID) and `syscall_number`—into underlying eBPF tracepoints for querying syscalls. While SQL-based observability systems today do not offer this functionality, we discuss a prototype implementation in §6.1. This data model allows the query engine to offer convenient, stable abstractions over OS functionality, our first two DSL requirements.

Verifier-Safety. Traditional SQL databases already transform queries into execution plans, guiding performance optimizations. In this case, the toolchain can ensure the execution plan is verifier-safe by moving operators between eBPF and user space. In Figure 2c, the plans start with two independent MVSFs attached to the `syscall_entry` and `syscall_exit` eBPF hooks, respectively. Both MVSFs send every `pid`, syscall id, and timestamp to user space via an eBPF map. The user space component matches entry and exit events, calculates the latency, and finds the maximum latency every second.

5 Optimizers for DSL-guided Iteration

DSLs with MVSFs guarantee verifier-safety, but the conservative MVSF-only execution plans seldom have good performance. To make extensions practical, DSLs must use an *optimizer* to produce execution plans that split functionality between eBPF and user space efficiently. Although other domains (*e.g.,* databases, program synthesis) use optimizers, the design and implementation of optimizers for eBPF that preserve verifier-safety remains an open problem. We discuss the

inadequacy of naïve approaches and present initial solution but also call for further research in this area.

5.1 Naïve Optimizers

A naïve optimizer could start with an execution plan that places the MVSF in eBPF and all other high-level operations in user space. Then, it could compose the first two operations in the program (*i.e.,* the MVSF and the subsequent operation) and generate the eBPF component of that plan. It could then repeat the process for the first three operations in the execution plan, and so forth. After this iteration, it could check each eBPF component against the verifier to find the execution plans that verify and use a simple heuristic (*e.g.,* largest eBPF component that passes the verifier) to choose the best one. Unfortunately, this strategy is impractical because it requires enumerating all candidate kernel-user space splits, which can result in a large search space of possible plans.

Another tempting strategy might preemptively terminate the search over execution plans based on some heuristic (*e.g.,* upon finding one that does not verify). Unfortunately, this strategy would be too conservative, since composing operations in eBPF might pass the verifier even if they fail separately. Importantly, neither approach accounts for secondary optimization objectives, which are critical for performance.

5.2 Research Opportunities

Instead, we envision optimizers that both account for secondary optimization objectives and preserve verifier-safety when guiding the toolchain's search. We call for the community to investigate the design of such optimizers, addressing the following open questions.

Cost models. A cost model estimates how well a candidate kernel-user space split maximizes a performance objective. This objective can vary depending on the domain. For example, observability queries might target low overhead and high throughput by estimating the overhead of running a query on an application. Meanwhile, congestion control and scheduling might target low latency, estimating the latency cost of executing parts of an algorithm in the kernel or user space. Finally, the cost model may need to capture different factors depending on which operators run in the kernel or user space.

Integrating with iteration. An optimizer integrated with DSL-guided iteration could use the cost and verifier model to inform the search process. Instead of naïvely enumerating every candidate execution plan, the optimizer could estimate the cost of a candidate before generating its eBPF code, prioritizing candidates with low costs. Depending on the domain, sophisticated optimizers could further transform a program's execution plan *during* iteration, reordering or replacing operations to generate a different but semantically equivalent program. Doing so might find better execution plans than were possible in the original plan.

Incorporating Verifier Modeling. In addition to modeling an execution plan's performance cost, the optimizer could also model the verifier's checks, as discussed previously in §3. For example, when considering possible kernel-user space splits, the optimizer could use program synthesis techniques to fuse operators the cost model indicates are expensive but the verifier model indicates will not verify. The cost model could even be embedded in the type system, as is the case in total languages [19, 32], and allow the compiler to prove that the program terminates within a certain number of instructions, a crucial constraint the verifier imposes.

6 Observability: A Case Study

Next, we extend the observability example from §3.2 to show how to achieve verifier-safety and performance with a toolchain that supports SQL queries for kernel observability, including a discussion of preliminary results.

6.1 Sketching a Kernel Observability Engine

To ensure that a toolchain compiles SQL queries to verifier-safe BPF programs while maintaining good performance, it needs a stable kernel schema, MVSFs, and an optimizer.

Stable schema. End-users write SQL queries using a stable schema that organizes the kernel's state and events into logical tables so that users can express data transformations (*e.g.,* filter, join, window) without kernel or eBPF expertise. Because the schema is a high-level abstraction over kernel objects, queries are portable across changes in the kernel, presenting a stable interface to the end-user. The stable schema would be maintained by the observability toolchain developers.

MVSFs. The tables and columns in the stable schema represent data accessible from specific eBPF hooks and helper functions. This limits the number of hooks that a generated eBPF program can reach and thus also the set of MVSFs. Since observability tasks only read data from the kernel, these MVSFs are small: they read the relevant data in the kernel using eBPF helper functions and immediately send the data to user space using an eBPF map. To guarantee that all the MVSFs are actually verifier-safe, the toolchain should include a test suite that exhaustively checks this for the kernel it is running in. When using MVSFs, the toolchain executes all other operations (*e.g.,* filters, joins) in user space. While MVSFs guarantee that a valid query will run, relying only on MVSFs can be slow, so executing MVSFs in the hotpath of key operations like system calls may introduce high probe effects to the system.

Optimizer. The eBPF programs that the toolchain generates from SQL queries should be fast and have low probe effect. While we leave the design of an optimizer for eBPF-based DSLs for future work, such an optimizer could borrow ideas from past work on databases (see §5).

For example, an optimizer could first transform SQL queries for observability into a logical plan with operators like filters

Implementation	Required Expertise OS	Required Expertise eBPF	Stable Interface	Verifier-Safe	Probe Effect (lower is better) Query 1	Probe Effect (lower is better) Query 2
Hand-Optimized	High	High	No	No	0.8%	5.0%
BPFTrace	Medium	Medium	No	No	1.0%	†6.1%
MVSF-only	Low	Low	Yes	Yes	72.7%	72.5%
Optimizer	Low	Low	Yes	Yes	1.2%	5.3%

Table 1: **Options for implementing observability queries. Query 1 and Query 2 columns show the probe effect on the application. † BPFTrace achieves poor query performance for Query 2, dropping 95% of events (though with low probe effect).**

and joins. Then, it could generate a physical execution plan including physical operations such as writing to eBPF maps and calling helper functions. During this process, it could perform logical optimizations (*e.g.,* predicate pushdown, join reordering) and physical ones (*e.g.,* batching, pipelining) to find a plan with a low estimated probe effect. The optimizer could use DSL-guided iteration to find verifier-safe eBPF programs and integrate the physical plan generation into the iteration process. The optimizer might use a simple heuristic which starts with an MVSF-only plan with all other operations in user space. The optimizer could then iteratively "move" operations from user space into the kernel and check whether the resulting eBPF program passes the verifier. Finally, the optimizer might estimate the probe effects of each such change and prioritize plans that have low probe effect. Of course, more advanced heuristics are possible. For example, we believe that some operations fit better in the kernel than others; while placing filters in the kernel is natural, SQL joins use unbounded loops and thus are incompatible with eBPF.

6.2 Preliminary Results

We performed experiments to evaluate the performance of MVSF-only extensions and whether a simple optimizer, coupled with DSL-guided iteration, can produce extensions with good performance. We consider two queries. Query 1 is the one shown in Figure 2c. It aggregates syscall information to find the maximum latency invocation of each system call, while Query 2 simply collects the latency of all system calls (*e.g.,* to correlate with other sources of granular observability data) and writes them to a file.

We evaluate four implementations for each query. The first implementation ("Hand-Optimized") represents a hand-optimized program an eBPF expert would write. For Query 1, it matches system call entry and exit using a shared eBPF map and then calculates the maximum latency using per-CPU maps. The user space program gathers latencies across all per-CPU maps and calculates the maximum latency per system call. For Query 2, it batches data in the kernel and sends that data to user space, which calculates the latency.

The second implementation is in BPFTrace [18]. BPFTrace offers a DSL with some abstractions for eBPF programming, but without verifier-safety or a data model. For Query 2, we configure BPFTrace to write system call latencies to a file.

The third implementation ("MVSF-only") uses an eBPF program containing only the MVSFs attached to eBPF's `sys_enter` and `sys_exit` hooks. Then, a user space program matches these events over time and calculates the latency of each captured system call. The fourth implementation ("Optimizer") is what DSL-guided iteration would produce: we implemented operators (*e.g.,* filter, project, join) and composed them to express the program. We executed each query to monitor a RocksDB application and report the difference in application throughput with each query and without (*i.e.,* the probe effect). We ran these experiments on a server running Ubuntu 22.04 (Linux v5.15) with two Xeon Gold 6150 CPUs (36× 2.7 GHz), 377 GiB RAM, and a Samsung 2TB NVM Drive.

Table 1 shows our results. As expected, "MVSF-only" performs poorly, with 73% probe effect on the RocksDB application. "Hand-Optimized" shows that it is possible to execute both queries efficiently, with 1% probe effect for Query 1 and 5% probe effect for Query 2. For both queries, BPFTrace introduces similarly low probe effects. However, BPFTrace requires eBPF expertise to write their programs since BPFTrace exposes eBPF maps and hooks, and its programs are not verifier-safe. In addition, BPFTrace drops 95% of the syscall events for Query 2 because its user space component cannot keep up with the data from the kernel. Finally, "Optimizer" achieves a probe effect that is comparable to "Hand-Optimized" and BPFTrace, but with a process that guarantees verifier-safety.

7 Conclusion

DSLs are a promising approach for kernel extensions, with the potential to provide an ergonomic interface, verifier-safety, and good performance. We call for further research on designing verifier-aware optimizations of DSL execution plans.

Acknowledgments

This work was supported by a gift from Intel, a Microsoft Grant for Customer Experience Innovation, an Amazon Research Award, and a Google Research Scholar Award.

References

[1] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. BBR: Congestion-Based Congestion Control. In *ACM Queue*.

[2] Zhongjie Chen, Qingkai Meng, ChonLam Lao, Yifan Liu, Fengyuan Ren, Minlan Yu, and Yang Zhou. 2025. eTran: Extensible Kernel Transport with eBPF. In *NSDI*. https://www.usenix.org/conference/nsdi25/presentation/chen-zhongjie

[3] Linux community. [n. d.]. sched_ext. ([n. d.]). https://github.com/sched-ext/scx Last accessed July 16, 2025.

[4] P4C community. [n. d.]. eBPF backend. ([n. d.]). https://p4lang.github.io/p4c/ebpf_backend.html Last accessed July 16, 2025.

[5] Jonathan Corbet. 2023. Reconsidering BPF ABI stability. (2023). https://lwn.net/Articles/921088/

[6] Mathieu Desnoyers and Michel Dagenais. 2008. LTTng: Tracing across execution layers, from the Hypervisor to user-space. In *Linux Symposium*.

[7] Kumar Kartikeya Dwivedi, Rishabh Iyer, and Sanidhya Kashyap. 2024. Fast, Flexible, and Practical Kernel Extensions. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. https://doi.org/10.1145/3694715.3695950

[8] The Linux Foundation. 2024. The State of eBPF. (2024).

[9] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. 2021. BMC: Accelerating memcached using safe in-kernel caching and pre-stack processing. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation*. 487–501.

[10] Prateesh Goyal, Akshay Narayan, Frank Cangialosi, Srinivas Narayana, Mohammad Alizadeh, and Hari Balakrishnan. 2022. Elasticity Detection: A Building Block for Internet Congestion Control. In *Proceedings of the 2022 Conference of the ACM Special Interest Group on Data Communication*. https://doi.org/10.1145/3544216.3544221

[11] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. 2021. ghOSt: Fast & Flexible User-Space Delegation of Linux Scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. https://doi.org/10.1145/3477132.3483542

[12] Jinghao Jia, Ruowen Qin, Milo Craun, Egor Lukiyanov, Ayush Bansal, Minh Phan, Michael V Le, Hubertus Franke, Hani Jamjoom, Tianyin Xu, et al. 2025. Rex: Closing the language-verifier gap with safe and usable kernel extensions. In *Proceedings of the 2025 USENIX Annual Technical Conference*. 325–342.

[13] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. 2021. Syrup: User-Defined Scheduling Across the Stack. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. https://doi.org/10.1145/3477132.3483548

[14] The kernel development community. [n. d.]. BPF Design Q & A. ([n. d.]). https://www.kernel.org/doc/html/latest/bpf/bpf_design_QA.html Last accessed January 4, 2025.

[15] Mohamed Khalfella. [n. d.]. Kernel crash when loading lttng-tracing module with IBT enabled. https://bugs.lttng.org/issues/1408. ([n. d.]). Last accessed May 23, 2025.

[16] 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 17th European Conference on Computer Systems*. https://doi.org/10.1145/3492321.3519562

[17] Martin KaFai Lau. 2020. bpf: tcp: Support tcp_congestion_ops in bpf. (2020). https://github.com/torvalds/linux/commit/0baf26b0fcd74bbfcef53c5d5e8bad2b99c8d0d2

[18] BPFTrace Maintainers. 2024. bpftrace. (2024). https://github.com/bpftrace/bpftrace Last accessed January 4, 2025.

[19] Dhall Maintainers. [n. d.]. The Dhall configuration language. ([n. d.]). https://dhall-lang.org Last accessed 22. May 2025.

[20] Ply Maintainers. 2024. Ply, a dynamic tracer for Linux. (2024). https://github.com/bpftrace/bpftrace Last accessed January 4, 2025.

[21] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Mauricio Vásquez Bernal, Yunsong Lu, and Jianwen Pi. 2019. Securing Linux with a faster and scalable iptables. *ACM SIGCOMM Computer Communication Review* 49, 3 (2019), 2–17.

[22] Sebastiano Miano, Fulvio Risso, Mauricio Vásquez Bernal, Matteo Bertrone, and Yunsong Lu. 2021. A framework for eBPF-based network functions in an era of microservices. *IEEE Transactions on Network and Service Management* 18, 1 (2021), 133–151.

[23] Sebastiano Miano, Alireza Sanaee, Fulvio Risso, Gábor Rétvári, and Gianni Antichi. 2022. Domain specific run time optimization for software data planes. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 1148–1164.

[24] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. 2018. Restructuring Endpoint Congestion Control. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*.

[25] Tomasz Osiński, Jan Palimąka, Mateusz Kossakowski, Frédéric Dang Tran, El-Fadel Bonfoh, and Halina Tarasiuk. 2022. A novel programmable software datapath for software-defined networking. In *Proceedings of the 18th International Conference on emerging Networking EXperiments and Technologies*. 245–260.

[26] osquery Maintainers. 2024. Welcome to osquery. (2024). https://osquery.readthedocs.io/en/stable/ Last accessed January 4, 2025.

[27] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. 2018. Floem: A Programming System for NIC-Accelerated Network Applications. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*. https://www.usenix.org/conference/osdi18/presentation/phothilimthana

[28] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and KK Ramakrishnan. 2022. Spright: Extracting the server from serverless computing! high-performance ebpf-based event-driven, shared-memory processing. In *Proceedings of the 2022 Conference of the ACM Special Interest Group on Data Communication*. 780–794.

[29] Athinagoras Skiadopoulos, Qian Li, Peter Kraft, Kostis Kaffes, Daniel Hong, Shana Mathew, David Bestor, Michael Cafarella, Vijay Gadepally, Goetz Graefe, et al. 2021. DBOS: a DBMS-oriented Operating System. In *Proceedings of the VLDB Endowment*, Vol. 15. VLDB Endowment, 21–30. https://doi.org/10.14778/3485450.3485454

[30] Robert Soulé, George Neville-Neil, Stelios Kasouridis, Alex Yuan, Avi Silberschatz, and Peter Alvaro. 2025. OSDB: Exposing the Operating System's Inner Database. In *The Biennial Conference on Innovative Data Systems Research*.

[31] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. 2021. Revisiting the open vswitch dataplane ten years later. In *Proceedings of the 2021 Conference of the ACM Special Interest Group on Data Communication*. 245–257.

[32] David A. Turner. 2004. Total Functional Programming. *Journal of Universal Computer Science* 10, 7 (2004), 751–768.

[33] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. 2023. Verifying the Verifier: eBPF Range Analysis Verification. In *Computer Aided Verification*, Constantin Enea and Akash Lal (Eds.). Springer Nature Switzerland, Cham, 226–251.

[34] 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 Conference*

of the ACM Special Interest Group on Data Communication. 50–64.

[35] Anil Yelam, Kan Wu, Zhiyuan Guo, Suli Yang, Rajath Shashidhara, Wei Xu, Stanko Novaković, Alex C Snoeren, and Kimberly Keeton. 2025. PageFlex: Flexible and Efficient User-space Delegation of Linux Paging Policies with eBPF. In *2025 USENIX Annual Technical Conference*. 291–306.

[36] Shawn Wanxiang Zhong, Jing Liu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2025. Revealing the Unstable Foundations of eBPF-Based Kernel Extensions. In *Proceedings of the 20th European Conference on Computer Systems*. 21–41.

[37] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, et al. 2022. XRP: In-Kernel Storage Functions with eBPF. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation*. 375–393.