Demystifying Performance of eBPF Network Applications

FARBOD SHAHINFAR, Politecnico di Milano, Italy
SEBASTIANO MIANO, Politecnico di Milano, Italy
AUROJIT PANDA, New York University, United States
GIANNI ANTICHI, Politecnico di Milano, Italy & Queen Mary University of London, United Kingdom

Recently, eBPF has emerged as the latest answer to how we should accelerate networked applications. In this paper we ask *can all networked applications benefit from eBPF?* We answer this question by running several benchmarks under different workloads and by designing different test cases. Our results show that in reality many networked applications cannot benefit from eBPF, and worse the use of eBPF can limit how applications are deployed (because eBPF can lead to performance isolation violations) and the workloads they can handle. We then discuss whether ongoing work can fix the limitations we identify, and propose directions that the community might want to focus on.

CCS Concepts: • **Networks → Programmable networks**; *Cloud computing*; • **Software and its engineering → Software performance**; *Operating systems*; • **Information systems → Enterprise applications**; • **General and reference → Measurement**.

Additional Key Words and Phrases: eBPF, Networked Applications, In Kernel Offload, Performance Measurement

ACM Reference Format:
Farbod Shahinfar, Sebastiano Miano, Aurojit Panda, and Gianni Antichi. 2025. Demystifying Performance of eBPF Network Applications. *Proc. ACM Netw.* 3, CoNEXT3, Article 16 (September 2025), 21 pages. https://doi.org/10.1145/3749216

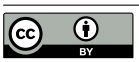
1 Introduction

Recently, eBPF has been widely used to implement network functions (NFs). This adoption is for three reasons: (a) it simplifies updating the logic for network functions such as OvS [73] that require in-kernel support because eBPF obviates the need to patch and update the kernel source code; (b) it allows NFs to achieve performance similar to those built using kernel-bypass libraries (e.g., DPDK [4]) while being easier to deploy [65]; and (c) its ability to modify kernel logic allows to improve legacy application performance (e.g., when connecting them through proxies) without compromising security or isolation guarantees [60]. eBPF has been used for many NFs including ones for load-balancing [65, 79], packet-switching [73], and accelerating protocols such as TURN [45].

There have also been recent moves to use eBPF to accelerate networked applications (rather than NFs) such as key-value stores [13, 25], consensus protocols [83], and transactional systems [84]. **Our goal in this paper is to understand eBPF's impact on networked applications.**

To see why this is important, consider Alice: she is building Yellow Sand, a new networked application that needs to go fast. She has read on the Internet that eBPF might help: she can move

Authors' Contact Information: Farbod Shahinfar, Politecnico di Milano, Italy, Milan; Sebastiano Miano, Politecnico di Milano, Italy, Milan; Aurojit Panda, New York University, United States, New York; Gianni Antichi, Politecnico di Milano, Italy and & Queen Mary University of London, United Kingdom, Milan.



ACM 2834-5509/2025/9-ART16
https://doi.org/10.1145/3749216

parts of the application logic into the kernel, and thus reduce latency and improve throughput. People tell her it might be a little painful, but unlike kernel modules, once she puts in the effort it will work on all versions of the kernel. However, before committing to eBPF she wants to know *if eBPF is actually going to help improve Yellow Sand's performance*? After all, these miracle cures for application performance don't always work. Faced with this problem Alice turns to the large body of literature on eBPF.

Unfortunately, this question is not directly answered by previously published papers (and blog posts). Existing work has instead focused on developing approaches where eBPF code can be used to customize many kernel subsystems [27, 30, 82], showing how some existing applications (*e.g.*, Memcached and consensus implementations) can be accelerated [13, 14, 25, 50, 83, 84]; and measuring the performance [30, 46, 58] of specific eBPF features (*e.g.*, maps or the cost of XDP). However, none of these explain the performance-and-complexity trade-off an application (*e.g.*, Yellow Sand) must navigate when using eBPF, nor do they explain the impact of running an eBPF accelerated application on other applications running on the same server.

A few months ago, we were faced with the same dilemma as Alice, which prompted us to write this paper. We used existing programs [25], and newly developed microbenchmarks[1] to answer four broad questions:

(a) Does offloading application logic to eBPF always improve performance (§3)? We find that this is not the case: Unsurprisingly, we found that whether or not application performance improves depends on how much of the logic can be offloaded. Perhaps more surprisingly, we also found that the workload and type of application significantly impact our answer, and that in many cases offloading to eBPF can result in no improvement, or even lead to worse performance.
(b) How do eBPF features, *e.g.*, collections and function chaining, affect application performance (§4)? Our analysis identifies how different usage patterns lead to different overheads.
(c) Does the use of eBPF affect other applications running on the same server (§5.1)? Surprisingly, we found that at present, one application's use of eBPF can affect the performance (both latency and throughput) of another, *i.e.*, applications using eBPF might violate performance isolation.
(d) Does the eBPF architecture, where code is compiled to a platform-independent bytecode and then compiled at load time in the kernel impose performance limits (§5.2)? We show that the current kernel JIT (which despite the name actually translates bytecode to machine instructions when the program is first loaded) produces suboptimal code, even for common code patterns (*e.g.*, `memcpy`), and we hypothesize that this lack of optimality cannot easily be addressed, since doing so would add significant complexity into the kernel.

Finally, we discuss (§6) how ongoing efforts within the community might affect our conclusions, and suggest areas that the community might want to focus on in the future.

2 Background

The life cycle of an eBPF program is as follows (see Figure 1). ① A user-space program requests to load an eBPF program into the kernel's memory through a system call. ② Kernel checks the program's safety properties using static analysis before loading it (more details can be found in §2.1). ③ If the program is safe, it is just-in-time (JIT) compiled to the host machine instruction and is linked with the API functions (more details can be found in Section 2.2). The JIT compiled program is now loaded inside the kernel memory. ④ the user-space program can attach the loaded eBPF program to a hook. A hook is part of the eBPF runtime that exposes specific events and provides the processing environment. ⑤ The arrival of events related to the hook will trigger the program (*e.g.*, a new input packet). ⑥ During its execution, the program can interact with the

[1] https://github.com/bpf-endeavor/bpf-app-offload-measurement

system using the API to perform predefined actions. Finally, the program may get detached and replaced with a new one, allowing users to update eBPF programs dynamically [23, 54].

2.1 eBPF Safety Guarantees

eBPF programs require verification because they run in the kernel space. Removing isolation will allow them to avoid costly operations such as context-switching. But the downside of running in such a high-privilege mode is that eBPF programs can interfere with the kernel outside of the expected API. This may lead to kernel crashes or violation of security policies. To insure an eBPF program will never attempt an unexpected operation, the verifier only accepts programs that it can prove their safe behavior. Memory access safety and program termination are the two major properties checked by the verifier [2, 12] (along others such as lock management).

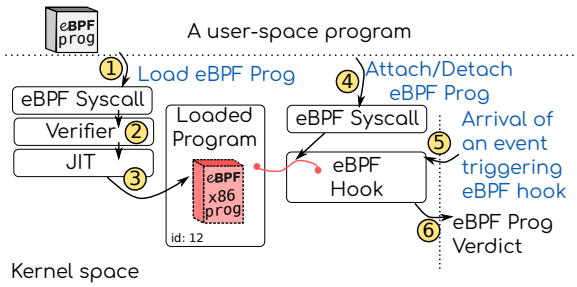


Fig. 1. Life cycle of an eBPF program

Memory Safety. The verifier checks that the program only reads from (or writes to) a memory address belonging to it. The verifier deduces this property by tracking types and values [74]. This technique can only automatically verify some memory accesses, and rely on the programmer for the rest especially when a variable is involved in defining the memory address. In these cases, the verifier requires the programmer to check the accessed memory address against a known bound. For example, when accessing a packet, they must check the address against the beginning and end of the packet. This check will create two execution paths. The verifier knows the program must not access the memory address on the path that the check fails. On the other, it is safe to proceed [12].

Program Termination. The verifier also proves the termination of the program. eBPF programs are not scheduled but run to completion on the same kernel thread triggered by the OS event they are attached to. A non-terminating program would stall the thread [12, 38]. Terminating the thread while maintaining the system's integrity is challenging [62]. By ensuring the termination of eBPF programs within a certain number of instructions, the kernel is confident that after invoking the program, it will release the resources eventually. In recent kernel versions, eBPF programs may be preempted [7]. Although preemption allows servicing the interrupts promptly, it does not help with the challenge of dynamically terminating an eBPF program.

2.2 eBPF Application Programming Interface

eBPF programs interface with the kernel by attaching to a hook provided by the eBPF runtime system and using its related helper functions. Below, we discuss some aspects of this interface related to this paper. First, we introduce eBPF MAPs that are used to preserve the state. Then, we introduce the eBPF networking hooks.

Memory Management Model. eBPF programs are invoked per each event, run-to-completion, and release all resources at their termination. Releasing the resources means eBPF programs can not maintain the application state across different invocations. To support stateful applications, an eBPF program can access a pre-allocated memory outside its execution environment through an

API called MAP. In this paper we refer to MAPs as "collections" to avoid confusion with hash-map data-structures and to be consistent with terminology of other programming languages[2].

There are different classes of collections, each exposing different memory access semantics. Array, hash-map, queue, and bloom filter are some of the examples. Some of the collections have a *per CPU* variant. These collections would expose a separate memory to each CPU core [46]. Since collections are allocated as part of the kernel memory, they can be shared with other eBPF or user-space programs to form communication channels [53]. Special collection types also exist for communication purposes (*i.e.*, ring).

eBPF Networking API. In this work, we focus on networking applications of eBPF and the hooks on the path of packets. Figure 2 highlights the placement of these hooks. The first hook on the receiving path of a packet is called `XDP` [31]. It provides access to packets directly at the NIC driver level, lowest-level possible, and enables efficient implementation of network functions [50]. Next hook is `TC` which is compatible with traffic control sub-system of Linux. It allows programs to process "socket buffers" (`SK_BUFF`) and use kernel networking services. For example, an eBPF program attached to this hook can broadcast a `SK_BUFF` and reduce the cost of this operation by avoiding multiple context-switches which is especially useful for implementing distributed applications [83]. Last hook, named `SK_SKB`, is located after the network stack (*i.e.*, TCP or UDP processing). It operates at the socket layer. On ingress path, it is triggered when a packet is placed into the receive queue of a chosen socket. This hook is useful for applications that require services from the network stack. For example, an eBPF based implementation of a database proxy uses this hook to reuse the existing TCP stack [14].

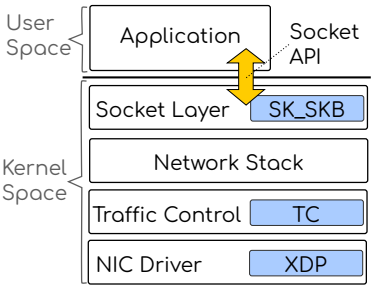


Fig. 2. Placement of eBPF hooks on network path.

3 eBPF Impact on Application Performance

This section explores trade-offs involved when offloading applications logic to the kernel with eBPF. We consider three cases where offloads can improve application performance: **(A) Full offload.** The eBPF program handles all incoming requests (§3.1). Katran [65], the production-grade L4 load balancer from Meta/Facebook, is an example of that. **(B) Fast-path offload.** The eBPF program acts as a fast path for processing some of the requests while others are sent to user-space (§3.2). For example, in the context of TURN server acceleration [45], the benefits have been shown when offloading only a specific type of request while leaving the rest to be handled by user-space. **(C) Pre-processing offload.** The eBPF program pre-processes all the received requests and sends them to user-space for further processing (§3.3). INTCollector [72] is an example for this case: INT telemetry data is pre-processed in eBPF and then sent to user-space.

Experimental setup. Experiments were performed using two servers connected back-to-back via 100 Gbps links through Mellanox ConnectX-6 NICs. Servers were equipped with the Intel Xeon Silver 4310 CPU running at 2.10GHz with 24 CPU cores. The CPU has 1.1 MB of L1 cache, 30 MB of L2 cache, and 36 MB of L3 cache. Servers have 128 GB of DDR4 memory with 3200 MHz frequency. Each machine has two NUMA nodes. Programs were configured to run on the same NUMA node connected to the NIC. We used Linux kernel version 6.8.0-rc7, version 14 of the `clang` for compiling the eBPF programs and targeted version 3 of the eBPF ISA [17]. Simultaneous multithreading

[2]Many programming languages and libraries provide support for frequently used data-structures such as arrays, linked-list, hash-maps under the name of "collections"

(hyperthreading) and Intel Turbo Boost were disabled during experiments to get consistent results. We also used this same setup for experiments discussed in other sections (§4, and §5).

3.1 Full-Offload: Move all logic to kernel

In this section, we look for an answer to the following question: *what type of programs can benefit from entirely executing in the kernel's eBPF runtime?*

Offloading a program to the kernel reduces the time spent between the wire and the processing logic. This, in principle, should lead to better latency. However, (1) overheads associated with the eBPF runtime and (2) the implementation details of the eBPF program itself must also be taken into account to better understand the potential benefits of a specific offload. Here, we focus only on the costs associated with the eBPF runtime and defer the analysis of program-specific trade-offs to §4.

The cost associated with the eBPF runtime can be divided into two components: the time to reach a hook and the time to prepare the eBPF environment. We measured the former using a modified NIC driver that timestamps received packets, and then take the time spent before the target hook is reached and eBPF environment is invoked. Hook overheads were, instead, taken by attaching a minimal program[3] and measuring the time to prepare the context object, enter the eBPF environment, and exit it. We averaged our tests over 100K invocations of the program itself and report them in Table 1.

Table 1. Time to reach an eBPF hook and prepare the execution environment. The origin of time measurement is the NIC driver. ± reports the interquartile range (IQR) value.

Hook	Time to reach (ns)	Prep. (ns)
XDP	0	38
TC	273 ± 51	35
SK_SKB	1072 ± 65	1350
Socket	5738 ± 779	-

Our results show that the time to reach a hook scales with the complexity of the OS network data path before the hook itself. The time to prepare the eBPF environment is, instead, hook-dependent.

We observed that preparing the eBPF context for XDP and TC hooks has little overhead (tens of nanoseconds). Surprisingly, the cost of preparing the context for SK_SKB is 35× higher than XDP and TC. Just attaching an eBPF program to this hook adds 1 μs to the processing time, which is the same order of magnitude as reaching user-space code (Table 1, socket). These overheads are because the kernel uses the sock_read and skb_read functions to get packets before invoking programs attached to SK_SKB, and these functions require more coordination[4] than the corresponding functions for the XDP and TC hooks.

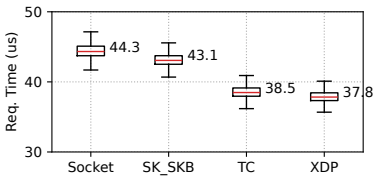


Fig. 3. Moving a complete path of application logic to the kernel can improve performance (latency). Figure shows an echo server offloaded to different eBPF hooks.

We analyze the effect of hook overheads on application performance using a simple echo server that we offloaded to all three hook points. By design, this application is I/O bound, and thus is most likely to benefit from offload. Figure 3 shows the response-latency distribution when the program is attached to each hook and run in user-space (shown under Socket). We observe that attaching the program to the XDP and TC hooks reduces response latency by about 6μs compared to a user-space application, but attaching it to the SK_SKB hook has nearly no impact on response latency.

[3]We attached a program that immediately exits.

[4]These functions operate on the socket queues. They acquire the bottom-half lock to dequeue the data.

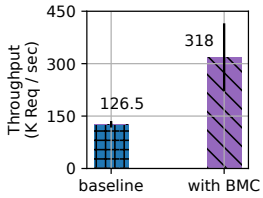


Fig. 4. BMC improves single-core throughput of Memcached by more than 2.5×.

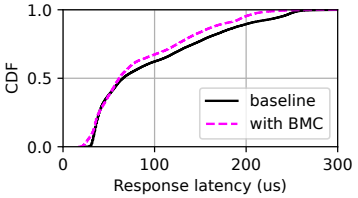


Fig. 5. Overall latency with and without BMC.

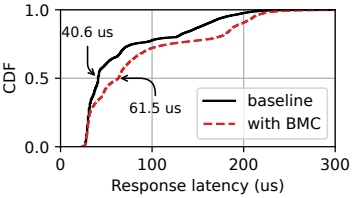


Fig. 6. Partial offloading of Memcached to eBPF/XDP degrades the performance of user-space traffic (while the fast-path enjoys performance improvement).

Our results show mixed benefits from offloading an entire application to the kernel's eBPF runtime: applications that operate on individual packets [6, 47, 49, 65] and can thus use the TC and XDP hooks might benefit from offload. However, applications that operate on streams or messages, and thus need to use the SK_SKB hook appear to have minimal improvements (for traffic arriving from outside the host). A further limitation reduces these benefits for all applications (regardless of the hook they use): eBPF has a limited ISA [71] that does not support many processor features including Single-Instruction-Multiple-Data (SIMD), cryptography offloads, floating point operations, etc. Consequently, offloaded programs cannot use these, and need to either use slower algorithms (*e.g.* non-SIMD equivalents, which can carry significant overhead: a non-SIMD version of xxHash that runs in the kernel is up to 7× slower than a user-space version [51]) or emulate missing operations (*e.g.*, floating point operations).

Thus, we conclude that full-offload only makes sense for simple packet-processing applications, such as load balancers and firewalls, which is also reflective of where eBPF is finding production adoption at present [60, 65]. Limitations of the current hook implementation and eBPF ISA make it so that offloading other application logic provides little or no additional performance benefits.

3.2 Fast-path offload: Move fast-path to kernel

Now that we have seen that full-offload does not improve the performance for most applications, we turn to the question of partial-offloads. Specifically, we consider the case of offloading the application's fast-path, and ask: *When does moving the application fast-path to the kernel improve performance?* Several recent works have proposed moving the fast-path of consensus protocols [83], database caches and proxies [14, 25], and WebRTC [45] implementations to eBPF, and process slow-path requests (*e.g.*, requests that access non-cached values, or trigger view changes in consensus protocols) in user-space. Thus, our question analyzes this popular approach.

Our analysis uses BMC [25], an eBPF program that accelerates Memcached key-value store by caching values in XDP. When a new request (*i.e.*, packet) arrives, it is handled by the eBPF program. Then, depending on the type, the program will answer directly or forward the request to user-space. This is the case, for example, of TCP requests, SET requests, or GET requests when the response value is larger than 1000 B. The BMC paper already reports its effect on Memcached response latency and throughput in a setting where most requests can be offloaded to the kernel, and shows that it can reduce latency by up to 10× and improve throughput by up to 16×. Therefore, we are interested in how fast-path offload affects overall application performance, specifically on its impact on requests that must be handled by the slow-path.

We first evaluate how BMC improves Memcached performance. We measure the throughput of Memcached with and without BMC using the Mutilate tool [1]. Our evaluation uses a single key

and value, ensuring that when BMC is used all requests are served from cache. We find (Figure 4) that in this case BMC increases the throughput of Memcached by more than 2.5×.

Next, we consider a more realistic scenario where some requests are served by BMC while others are served by Memcached in user-space. In this case, we deployed BMC on a server and coupled it with a Memcached instance with four worker processes. We populate Memcached with two key-value pairs: a small value that can be cached by BMC (and accesses which is accelerated) and a large one that can only be processed in user-space. We generated background traffic with the rate of $50K$ requests per second for the small value that takes advantage of the BMC fast path. Alongside it, we sent a set of requests for the large value that must be processed in user-space ($5K$ requests/sec). We selected a relatively low load for both flows to investigate the performance interference in a realistic setup with modest contention.

Figure 5 shows BMC improving the overall response latency of flows as expected. But, if we consider the flow going to the user-space in isolation, we notice that they are experiencing stall. Figure 6 shows measured values for large requests and compares it to the response latency when BMC is not used: we find that slow-path requests experience a 51% increase in median latency ($\sim 20.9\mu s$), and an 11% increase in tail latency ($\sim 24.4\mu s$ at 99^{th} percentile). This is because slow-path requests must be processed by BMC's kernel code before being forwarded to the user-space. This increases the amount of processing and, thus, service time for slow-path requests, resulting in increased latency.

Thus, we conclude that the benefits of fast-path offload depend on the workload characteristic. If most requests can be handled by the fast-path, one can achieve the latency and throughput benefits reported by the BMC paper. However, if a significant fraction of requests need to be handled by the slow-path then offloading the fast-path to the kernel's eBPF runtime can result in performance degradation. This suggests that fast-path offload approaches might benefit from a dynamic scheduling policy, where their use depends on the observed workload.

Here is a formulation for quantifying the fraction of requests that fast-path must handle so that the system benefits from partial offload. Assume we are offloading to eBPF/XDP and *the eBPF program is the first component that runs*. Let C be the cost of processing a request before offload. Let ρ denote cost of eBPF program as a percentage of C. Finally ω will represent the percentage of requests belonging to the fast-path. Given this definitions, cost of fast path is ρC and cost of slow path is $C + \rho C$. As a result the expected cost of processing a request after offloading will be the weighted sum of each path ($\omega\rho C + (1-\omega)(1+\rho)C$). We want this new cost to be less than the cost before offloading (C). We see this happens only when the percentage of request for the fast path (ω) be more than ρ: $\omega\rho C + (1-\omega)(1+\rho)C < C \implies \rho < \omega$

Fast-path offload in case of kernel bypass. Recently, AF_XDP has emerged as an alternate approach to using eBPF to accelerate networked programs. Programs in user-space can establish AF_XDP sockets (XSKs) on which they receive (or send packets) that bypass the kernel network stack. When using XSKs, programs provide eBPF logic that runs in the kernel and is responsible for determining what received packets should bypass the network stack and be delivered to the user-space. The use of AF_XDP can reduce the overheads for requests sent to the user-space.

We evaluated how BMC can improve XSKs performance using a key-value store that implements the Memcached protocol but uses XSKs for network I/O. We provide the BMC logic as a part of establishing the XSK: requests that cannot be serviced from the BMC's cache are forwarded to the key-value store. This closely resembles the setup above but uses XSKs.

Our results (Figure 7), shows that BMC with AF_XDP yields more modest performance improvements because an AF_XDP based key-value store has higher performance than Memcached.

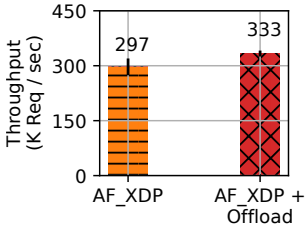


Fig. 7. Fast-path offload improves performance of the key-value store application running on top of zero-copy AF_XDP sockets.

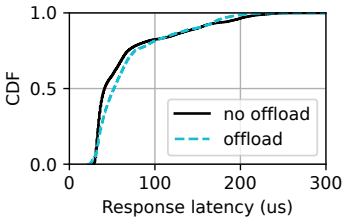


Fig. 8. Overall latency with and without partial offload.

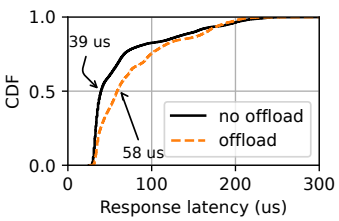


Fig. 9. Partial offloading affects the latency of flows going to the user-space through an AF_XDP socket.

Specifically, we observed a 12% throughput improvement from using BMC's partial offload, compared to the case where all application logic was run in user-space. Furthermore, in this case we also observed no discernible latency improvements (Figure 8), because bypassing the network stack already yield low-latency. Finally, if we focus on requests processed in user space (Figure 9) we observe that BMC does affect the median latency of these requests by increasing the processing time in the kernel. Thus we conclude that offloading the fast-path is even less useful for AF_XDP based applications, because it cannot significantly improve response latency nor reduce the number of cycled required to process requests.

3.3 Pre-processing offload: Pre-process in kernel

Another deployment mode for eBPF offloads is the one where the in-kernel program modifies data before delivering it to the application in user-space. In this section, we thus look for an answer to the following question: *does pre-processing packets in the kernel on their path to user-space provide performance advantages?*

Here, we focus on throughput rather than latency. This is because all packets in this setting are delivered to user-space, and thus experience worse latency due to the overheads of eBPF context preparation (§3.1) and executing additional logic in the kernel (§3.2).

Table 2. Attaching a minimal eBPF program to datapath increases the in-kernel processing and decreases bandwidth measured by `iperf`.

Scenario	Bandwidth (Gbps)
No eBPF	7.24 ± 0.64
XDP	7.14 ± 0.22
TC	7.02 ± 0.89
SK_SKB	6.22 ± 0.30

We start our analysis by measuring the change in throughput when placing a minimal eBPF program on the path of packets to the user-space. This allows us to understand overheads associated with the eBPF infrastructure. Table 2 shows how the bandwidth measured by `iperf` (version 2.1.5) changes when the eBPF program is attached to different hooks. The experiment consists of 48 parallel TCP connections going through a single hardware queue (*i.e.*, single core for eBPF) to put maximum pressure on eBPF program. The experiment lasted 10 seconds, and it was repeated 100 times. Noteworthy is that `iperf` reported 3% less bandwidth after attaching the eBPF program to TC and 14% less when it was attached to SK_SKB.

Now that we have a better understanding of the overheads coming from eBPF, we can start exploring what type of pre-processing can provide performance benefits. First, it is worth noting that any type of pre-computation (*e.g.*, checksum or header parsing) would unlikely bring performance

advantages as it is anyway performed on the same CPU as for the user-space program. What can really help is to reduce the cost of transitioning to user-space. This reduction could be in the form of kernel bypass (*e.g.*, using `AF_XDP` [39]) or by acting directly on the size of the packets being moved through the OS stack: the insight is that reducing this data leads to smaller memory-copy from kernel to user-space and better performance.

Here, an eBPF program can potentially reduce data movement by pre-processing requests and sending less data to user-space. To better understand this, we generated maximum-size UDP messages (1458 B) toward a socket application. We attached an eBPF program to `XDP`, which truncates the messages, reducing data sent to the user-space program. The rate of processing requests was measured by varying the amount of data reduction. We show the results in Figure 10: reducing the payload size in `XDP` resulted in only modest performance improvement (at most 2.8% or 27 kpps).

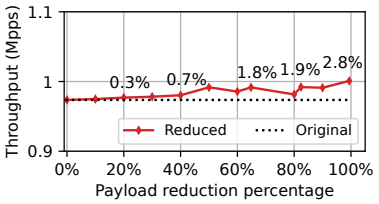


Fig. 10. Reducing the packet size before copying to user-space does not increase the throughput significantly.

We note that this is the best-case scenario for performance improvements: the tested eBPF program just blindly truncates packets without any specific logic. If more refined processing needs to be performed, then overheads associated with the eBPF APIs can come into play, further decreasing potential performance benefits. Noteworthy that in this test, we used the standard socket API interface to pass packets from kernel to user-space. eBPF provides different communication channels for this task that we analyze in §4.

Thus, we conclude that it is unlikely that pre-processing data in the kernel with eBPF can significantly improve application performance.

4 Implementation Cost in eBPF Programs

In this section, we analyze costs associated with the implementation details of eBPF programs. We start our study by benchmarking eBPF collections (§4.1) as they are widely used, especially in the context of stateful applications. We then turn our attention to the costs associated with eBPF sandboxing. In particular, we report our findings on how increased program complexity can affect the throughput (§4.2). Finally, as multiple programs can be connected together so to avoid the complexity limits imposed by the verifier, we focus our attention on the overheads imposed by mechanisms for program chaining (§4.3).

4.1 Collections

In this section, we look for an answer to the following question: *What are the overheads and trade-offs associated with eBPF collections?* Collections are primarily used for two purposes by eBPF programs: (A) to store internal state; and (B) to communicate with the user-space. We analyze the overheads for each use case below.

Table 3. Time to access a collection from eBPF program. Values are averaged over 100K measurements. A 4-byte key was used for all cases.

Memory Location	Access Time (ns)
Array (`BPF_MAP_TYPE_ARRAY`)	3.9 ± 0.8
Hash-map (`BPF_MAP_TYPE_HASH`)	18.8 ± 1.2
LRU hash-map (`BPF_MAP_TYPE_LRU_HASH`)	19.2 ± 1.4

Collections for internal state. We start by measuring access latency and throughput for collections used to store internal

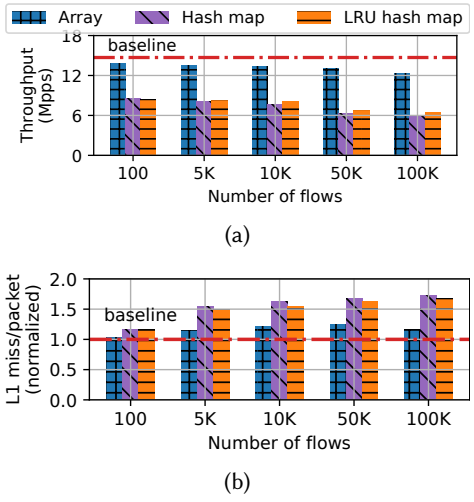


Fig. 11. (a) Memory footprint of eBPF maps affects their performance. (b) This is correlated with the increase in the data cache misses.

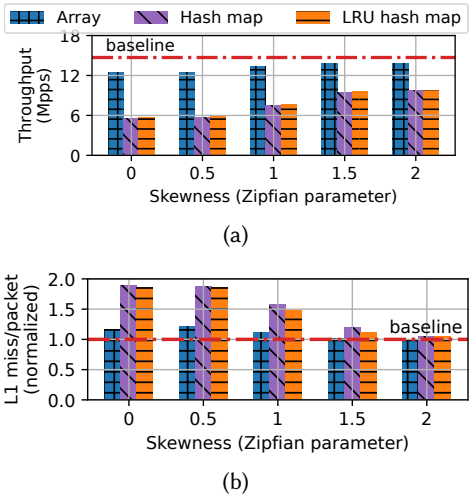


Fig. 12. (a) Access pattern to eBPF maps affects their performance. (b) This is correlated with the increase in the data cache misses.

state. Building on prior work [46], we use 4 B keys for all our tests, because this minimizes key-processing overheads. Table 3 reports access times for three common collection types: arrays, hash-maps, and LRU hash-maps. LRU hash-maps are a fixed-size data structure that evicts least-recently used entries to make space. As is expected, we observe that arrays have lower access latencies than either hash-map variant.

To read a value from a collection, eBPF programs use a helper function that calls the collection-specific routine. As a result, one access to a collection will require two function calls. The eBPF JIT optimizes accesses when possible by inlining (replacing) the call to the helper function. For arrays, this optimization allows the appropriate memory to be directly indexed by the program, improving access time by 20%. For the hash-map and LRU hash-map, the call to the helper function is replaced with the direct call to the appropriate routine, improving their access time by 8% and 4%, respectively.

Next, we focus on how much the memory footprint of eBPF collections as well as the access patterns impact performance. To do so, we wrote an `XDP` program that keeps per-flow packet counters. For hash-map and LRU hash-map, we indexed using the standard five-tuple, and for the array, we combined the source and destination port to compute an index into the array. We then ran an experiment where we varied the number of flows and configured the size of each collection so that up to 90% of the collection's capacity was used. We needed to limit occupancy to 90% of the capacity to allow us to fairly compare LRU hash-map performance to that of a standard hash-map. Figure 11a shows the result of this experiment when we feed as input traffic 64 B UDP packets. The marked baseline refers to a `XDP` program that only parses the packet without performing any access to the collections.

We observed that the access overheads depend on collection type and increase as the size of the collection grows: compared to a program that accesses 100 flows, throughput for a run with 100K flows is 10.8% lower when using an array, 30.5% lower when using a hash-map, and 22.6% lower when using a LRU hash-map. We hypothesize that this reduction is due to an increase in working set size and checked this hypothesis by measuring the cache miss rates (Figure 11b). We found that

Table 4. Time to retrieve an item from an eBPF collection from user-space. Values are averages of 100K look-up operation. A 4-byte key was used. For Ring, 100K events where enqueued before timing the accesses.

Collection Type	Access Time (ns)
Array	485
Hash map	495
LRU hash map	495
Array [per CPU]	1552
Hash map [per CPU]	1550
LRU hash map [per CPU]	1554
Ring queue	10
Array [MMap]	2
AF_XDP RX Ring	14

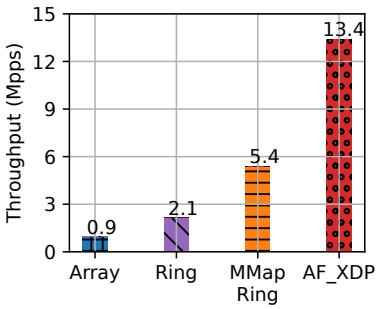


Fig. 13. Maximum throughput achieved when using different implementations for sending data to user-space.

the miss rate increases as we increase the number of flows, and an increase in working set size is indeed responsible for the observed overheads.

But cache miss rates depend on access pattern, and uniform flow distributions are rare in practice. Thus, our next experiment aimed to understand how other flow distributions affected throughput. To do so, we configured the load generator to choose among 100K flows using a Zipfian distribution. We varied the Zipfian parameter from 0 (a uniform distribution) to 2 (a skewed distribution)[5]. Figure 12 shows the throughput and cache miss rate as a function of the Zipfian parameter. We observe that throughput increases as we use a more skewed distribution, showing the importance of increased cache locality.

We conclude that eBPF collections, when used to store internal state, have similar performance trade-offs as in other environments (*i.e.*, arrays accesses are faster than hash-map accesses). More importantly, we find that the use of collections can reduce throughput by up to 30% in the presence of hundred thousands of flows. This indicates that similar to other throughput or latency sensitive programs, developers writing eBPF programs must carefully consider collection size and access pattern to minimize performance overheads.

Collections for communication with user-space. Next, we look at the cost of using eBPF collections to exchange data with applications in user-space and compared it when using `AF_XDP`. Different types of collections can be used for this purpose: (1) array and hash-maps. Katran [65], a layer-four load-balancer, uses these for control plane—data plane communication: the control plane located in the user-space uses hash-maps for configuring the IP mappings in the eBPF data plane. (2) per CPU variant of array and hash-maps that use a different memory region for each CPU. These can be used in a similar manner as arrays and hash-maps, but avoid synchronization. (3) ring queues. These are used by INTCollector [72], a program that processes telemetry packets in the kernel and then uses a ring queue to send the extracted data to a user-space database.

Table 4 shows the access time from user-space for each of these three eBPF collection types. We measured these times using a benchmark where we store a single 64 B entry in each collection and access it repeatedly (100K) from user-space. We measured the duration of performing this experiment and divided it by the number of accesses to report the average access time. When comparing the results to the measured access time from the kernel (Table 3), we note that latency increases by up to two orders of magnitude. This is because reading array or hash-maps from

[5]These parameters match observations from Twitter's object cache workload, where accesses followa Zipfian distribution with parameter between 1 to 2.5 [78]

user-space requires a system call. This cost is higher for their *per CPU* variants because more data (one value for each core) is copied from kernel to user-space[6]. Rings and memory-mapped arrays do not suffer from the same overheads because similar to AF_XDP rings [39], they do not impose syscall overheads.

Next, we measured the throughput of different data structures when communicating between eBPF and a user-space program. In particular, we compared the ring collection, which is the standard mechanism provided by eBPF, to using an eBPF array for bulk data communication. To communicate using an array, the user-space application and eBPF program must share a head and tail index, and we used a memory region accessible from both kernel and user-space for this. We evaluated throughput by using both approaches by having an external traffic generator send 64 B packets (to avoid memory copy overheads that we discuss in §5.2). The eBPF application copies payload into the data structure, from which the user-space application will read. We show results for both (Array and Ring) in Figure 13. As expected, we find that the ring offers higher throughput because it allows data to be accessed without a syscall. However, we were surprised by the low throughput (2.1Mpps) achieved by the ring: we think this is because when no data is available, the user-space application would block rather than busy polling when accessing the ring.

To test this hypothesis, we developed a second data structure (MMap Ring) where a ring buffer (including both metadata and data) is directly accessible from both user and kernel space. We designed this data structure so that user-space applications will busy-poll when no data is available in the ring. We repeated the same experiment, and our results (MMap Ring in Figure 13) show that MMap ring has 2.5× higher throughput. However, MMap ring is potentially slower than AF_XDP sockets (XSK): unlike XSK, the MMap ring requires data copies. We evaluated this by measuring throughput for redirecting packets over an XSK, and Figure 13 (AF_XDP) shows that this achieves even 2.48× higher throughput than MMap ring. But XSKs cannot trivially be used for all kernel-userspace communication. Thus we observe that avoiding copies is useful, but is only possible in a few cases.

4.2 Program Complexity

eBPF is used for various use cases ranging from small programs checking the header of packets to more complex programs that operate on application layer protocols. In this section, we ask *how much sandboxing costs affect the throughput of eBPF programs when their complexity increases?*

To find the answer, we prepared an experiment comparing the performance of a packet processing program when running inside eBPF (`XDP`) against when it is hard-coded in the NIC driver. In the latter case, the program was called[7] at the same place where the driver would invoke the `XDP` hook, and it received the same input object as `XDP`. We considered two programs: one compute heavy that calculates Fibonacci numbers, and one that is memory intensive that sums the first 64 B of the received packet repeatedly.

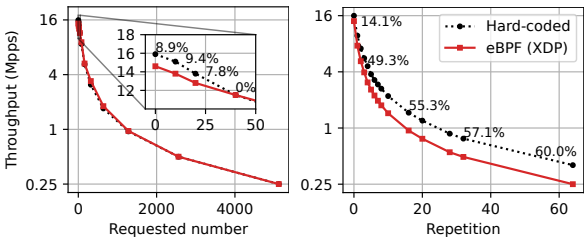
Figure 14a shows the result. We see that as computational intensity increases, the throughput of the eBPF program becomes the same as the hard-coded program. This shows that the eBPF implementation is as efficient as the native implementation, and that the cost of sandboxing is minimal (8.9% observed with at the X=0) for compute heavy tasks.

Surprisingly, Figure 14b, which depicts the result for the program calculating the summation of the first 64 B of the packet, shows a different behavior. Looking at the machine instructions generated for each program, we noticed that the hard-coded version spends 326 instructions in

[6]In this experiment we used 24 cores. The cost will be higher for machines with more cores.
[7]The control flow of driver code was directed toward the program code.

each iteration of summation[8], but the eBPF program spends 520 instructions[9]. Specifically, we find that the compiled eBPF program is more conservative with the use of registers and this is due to the limited number of registers defined in the eBPF ISA. Indeed, eBPF specifies 10 general purpose registers compared to x86, which has 16.

Our takeaway is that for sufficiently complex programs, the sandboxing costs of eBPF can become insignificant. As a consequence, eBPF can potentially run as fast as native programs given they have compiled to the same machine instructions. However, current limitations in the eBPF ISA can impose constraints on its compilation and, as a result, generate less optimized code.



(a) Calculating n^{th} Fibonacci number. Throughput of this compute-heavy program converges when the complexity increases.

(b) Summing the first 64-B of packet. Throughput of this branch-heavy program diverges due to poor code generation of eBPF JIT.

Fig. 14. Comparing the code generation quality of the JIT-compiler against a native compiler (clang). Both process the same C program, so higher throughput means more efficient compilation. The reported percentages are the difference of native case relative to eBPF.

4.3 Program Chaining

In eBPF, program chaining is particularly helpful to overcome the complexity limits imposed by the verifier. In fact, by decomposing an application logic into multiple parts (called sub-programs), each will be less complex and more likely to satisfy the verifier requirements as it will be treated separately. In this section, we look for an answer to the following question: *How much overhead does the eBPF infrastructure add when chaining programs?*

To the best of our knowledge, there are three main mechanisms provided by eBPF to support program chaining: (A) tail calls that have been studied in the past [16, 37]; (B) the BPF-to-BPF [66] function call; and (C) BPF global functions [68] that allow a program to call another program similarly to a function call via a trampoline.

eBPF tail-calls allow programs to be structured as data flow graphs, similar to Click [42], allowing packets (or data) to be passed from one eBPF component to another. Tail calls are implemented by performing a lookup on a collection of pointers to other eBPF programs, which introduces overhead associated with indirect jumps into the program. This overhead has significantly become more costly after discovering speculative vulnerabilities such as spectre [37, 41]. Recently, eBPF's JIT compiler has sought to mitigate these problems by optimizing tail-calls when the next program in the chain can be determined through static analysis.

We evaluated the cost of each of the three program chaining mechanisms and the effect of the recent JIT changes using BMC. BMC's current implementation makes three or more tail-calls on its fast-path, depending on the number of requested keys in a query. We compared five different versions of the program: (1) The original one; (2) A version that does not benefit from the JIT compiler optimization discussed above; (3) A version without tail-calls: we replaced the tail-calls with calls to functions that were marked to be inlined by the compiler. No further changes were made to the code, although more optimizations might have been possible, such as avoiding map lookups used for coordination between programs. To load the program into the kernel, we increased

[8]Five instructions per each byte copied (64 times) and six instructions for repeating the operation (X-axis of Figure 14b)
[9]Eight instructions per each byte copied (64 times) and eight instructions for repeating the operation (X-axis of Figure 14b)

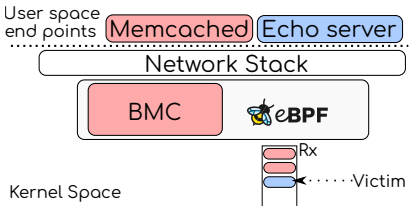


Fig. 15. eBPF programs may interfere with the performance of other networked programs. The blue packet will experience a longer queuing time because the red packets will have a longer processing time.

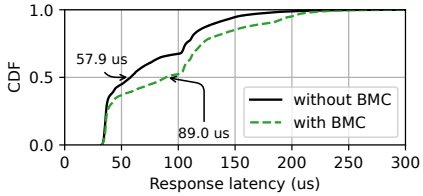


Fig. 16. Offloading Memcached to eBPF (XDP) increases the request completion time of another network application.

the complexity limit inside the eBPF verifier from one million instructions to ten million; (4) A version that uses *BPF-to-BPF* calls instead of the tail-calls; (5) A version that uses *BPF global functions*.

We generated requests using Mutilate [1], each one querying a single key so that we deterministically know that the original BMC fast-path encounters three tail-calls.

Table 5. Removing tail-calls improve the performance by 6.5% (1.7% with JIT optimization for constant keys).

Scenario	Throughput (M Req/Sec)	Improvement (%)
JIT optimization disabled	1.07	0%
JIT optimization enabled	1.12	4.7%
No Tail-calls	1.14	6.5%
BPF-to-BPF	1.11	3.7%
BPF global func	1.14	6.5%

Table 5 shows the result of this experiment. Using tail-calls with JIT optimization enabled imposes little overhead: when compared to the case of not having tail-calls at all, the program suffers only 1.7% throughput degradation. This shows that the JIT optimization is effective in reducing the cost of jumping between programs. Indeed, if they are disabled, the program suffers a possible performance penalty of up to 6.5%. BPF-to-BPF allows for a very similar performance of enabling JIT optimization with the tail-calls mechanism. Interestingly, the eBPF global function seems to completely remove penalties associated with program chaining, making this option very attractive for complex programs requiring multiple sub-programs to pass the verifier checks.

5 Other Concerns when Offloading to eBPF

During our experiments we found that using eBPF offloads can have unexpected impact on the performance of other applications (§5.1), and on the performance of the application whose logic is being offloaded (§5.2). The former is because eBPF can violate performance isolation, while the latter is because the current JIT does not consider the processor architecture. We discuss both issues in this section.

5.1 Lack of Performance Isolation

During our experiments, we observed that offloading one application's logic could affect the performance of another. To better analyze this, we setup the scenario shown in Figure 15: We run two applications, Memcached with the BMC offload, and an echo server. We configured the NIC so that the same network queue was used by both applications. This is realistic because most NICs have a small number of queues and the default configuration for most servers uses a single queue for each core [26, 28], and uses RSS [9, 61] to randomly assign packets to cores. Thus, packets from two different applications are likely to end up in the same queue.

Because the echo server does not use an eBPF offload, we expect that its performance is not affected by the BMC offload. Our experiment aims to check if this is the case. To do so we sent requests to both applications, and measured the echo application's response latency. Our results, in Figure 16 shows response latency when the echo application runs by itself (without BMC) and in the scenario described above. Surprisingly, we find that median response latency increases by 45%. This is because the BMC logic is executed before packets are demultiplexed, and thus packets destined for the echo server are delayed until earlier packets (for Memcached) have been processed.

Our observations shows that eBPF offloads can lead to performance isolation violations. Unfortunately, fixing these violations presents a trade-off: the kernel can delay eBPF offload execution until after demultiplexing, but this would reduce the performance benefits of offloading. Thus, administrators must take care when collocating an application that uses eBPF offloads with one that does not to avoid performance interference, *e.g.*, by using mechanisms like receive flow steering (RFS) [28] to map applications to different queues.

5.2 Architectural Blindness

When loading an eBPF program, the kernel's JIT[10] translates the program into machine code that can be efficiently executed. Unfortunately, the JIT [21, 23, 64] generally translates each instruction into the corresponding machine code. This simplicity reduces the amount of code required in the kernel, helping maintainability. However, it also leads to a situation where the compiler cannot emit more efficient code.

Table 6. Copying 1 KB of data inside the kernel. eBPF imposes a 10× overhead with respect to a kernel module as it does not use string instructions such as `rep movs` from the x86 ISA.

Scenario	Time (ns)
kernel module	32 ± 0.7
eBPF	340 ± 1.8
eBPF with `memcpy` wrapper	36 ± 1.4

We demonstrate this by using a program to benchmark `memcpy`. It creates a $100 \times 1KB$ element array and copies a $1KB$ object to each array element. We measure the time to run the entire test and divided it by 100 so to obtain the average time for a single `memcpy` operation.

Table 6 shows time taken for each operation when the benchmark is implemented as a kernel module and in eBPF. Surprisingly, the eBPF program is about 10× slower than the native code. To understand why this was the case, we dumped the resulting instructions for each. The native code used `rep movs`, Intel's recommended approach for implementing `memcpy` [33]. On the other hand, the eBPF JIT produces a loop that copies word-by-word, leading to the observed performance difference. To further confirm that this difference was responsible for the observed performance difference, we created a `memcpy` `kfunc` [40] that used `rep movs` and could be called from eBPF. The 'eBPF with `memcpy` wrapper' row in Table 6 shows that the `kfunc` has performance comparable to the native kernel module.

Our observation shows another trade-off that the kernel and the eBPF ecosystem needs to traverse: a better kernel JIT can significantly improve eBPF performance, but would require incorporating parts of a compiler into the kernel. Similarly, a larger set of `kfuncs` can address the kernel JIT's limitations, but at the cost of additional kernel code and complexity. The community must thus choose between its desire to limit kernel complexity and its desire to move performance sensitive application code into eBPF.

[10]The JIT in this case is a misnomer: eBPF programs are compiled on load, *i.e.*, the kernel uses an ahead-of-time compiler, rather than compiled while being executed.

Table 7. A summary of some of the key observations of this paper.

Section	Takeaway	Compiler	Verifier	Runtime	ISA	External functions
§3.1	Offloading to eBPF reduces the time to process the packet	–	–	✓	–	–
§3.2	Fast path of eBPF can interfere with the slow-path	–	–	✓	–	–
§3.3	Preprocessing packets in the kernel is not very effective	–	–	–	–	–
§4.1	eBPF maps are efficient, but the access pattern matters	–	–	✓	–	–
§4.2	eBPF programs may not compile to the optimal native machine code	✓	✓	–	✓	–
§4.3	Chaining eBPF programs with BPF global func brings little overheads	–	–	–	–	–
§5.1	Programs may interfere with other network applications on the system	–	–	✓	–	–
§5.2	Programs do not benefit from all the features of their host processor	✓	–	–	✓	✓

6 Discussion

eBPF is evolving rapidly. New hooks have been introduced recently so that eBPF programs can customize more subsystems including scheduling [27], storage [82] and more [18, 80]. At the same time, verifier and JIT improvements have eliminated expressibility limitations on loops [19] and pointers [43], to name a few. Furthermore, the academic community and industry continue to propose changes that are meant to improve the performance and usability of eBPF programs [22, 57, 76]. Given this, in this section we address the question *how might changes in the eBPF ecosystem affect our findings?* This question is important for two reasons: (a) it allows the reader to gauge the longevity of our findings; and (b) more importantly, it allows the community to determine what areas to focus on.

We structure our discussion in terms of eBPF components: (a) runtime; (b) compiler; (c) eBPF ISA; (d) verifier; and (e) external functions. Table 7 provides an overview of how each of these components affect the problems described in the previous sections.

Runtime. Improvements to the eBPF runtime can fix a majority of the problems we identified in previous sections. We use the term "eBPF runtime" to refer to the kernel code that is responsible for defining hooks, running an eBPF program when a hook is encountered, and for providing the APIs that this program can use. For example, the lack of performance isolation (§5.1) and interference between the fast- and slow-path in partial offload scenarios (§3.2) can be solved by implementing scheduling mechanisms or adding buffers that allow eBPF hook execution to be deferred [67]. Similarly, one can improve access times for eBPF maps (§4.1) by having the runtime intelligently prefetch values to reduce the impact of cache misses. Similarly, the runtime can batch calls to the same eBPF program to further improve cache locality.

Thus we conclude that runtime changes are the most likely to fix the problems that we discussed, however some of these changes, *e.g.*, prefetching, batching, and deferred calls, require additional program information (*e.g.*, to determine what values to prefetch) or significant changes to the kernel (*e.g.*, to add the ability to buffer packets and delay invoking an eBPF program).

Compiler and JIT. In two cases (§4.2 and §5.2), we observed bad performance because the compiler and eBPF JIT generated suboptimal machine code. Clearly, improvements to one or the other of

these components would address this problem. We discussed what improvements are necessary, and the challenges they pose previously in §5.2.

eBPF ISA. Related to the previous point, the current eBPF ISA [71] might also limit the compiler's ability to generate optimized code. Unlike JVM (which models a stack machine) and LLVM (which models a machine with infinite registers), the eBPF ISA models a machine that is closer to real hardware, providing a mixture of registers and stack memory. However, the eBPF ISA offers many fewer registers than existing machines: eBPF defines 10 registers while the AMD64 ISA provides 16 general purpose registers and several more SSE registers. A smaller set of registers produces a harder register scheduling problem, limiting the compiler's ability to generate efficient code. Consequently, we believe evolving the eBPF ISA might yield significant gains. Recent attempts to standardize the ISA by producing an RFC might both aid in improvement efforts (by providing a standard target to evolve) and hinder it (by adding inertia).

Verifier. As other have observed [24, 76], the eBPF verifier limits expressivity, which in turn makes it hard to provide better abstractions to users. For example, verifier limitations force programmers to add needless bounds checks in some cases, and prevent the compiler from removing these checks. Thus improvements to the verifier make it easier for programmers to writer performant eBPF code, including code that might avoid the performance anomalies we identified in this paper. Further, similar to ISA improvements, verifier improvements might also make it easier to adopt better compiler optimization passes, further improving performance. Fortunately, many in the community are already focused on improving the verifier [3], but we are unlikely to benefit from these opportunities without focus on the other components in the eBPF ecosystem.

External functions. eBPF programs can call an ever growing set [81] of external functions (or `kfuncs`). These functions are written in native code, and can thus take advantage of architecture specific instructions and are not subject to verifier limitations. As a result, these functions often have better performance than the eBPF equivalent, and help improve the performance of eBPF offloads. For instance, in Table 6, we showed that a `memcpy` `kfunc` could significantly outperform an eBPF implementation. Fortunately, programmers can already add `kfuncs` by creating a kernel module, but it is far from clear what `kfuncs` should be added (as a reminder, most kernel modules in Linux are included in the kernel tree). The choice of `kfuncs` can both improve performance and enable new use cases, and thus the community should identify `kfunks` that implement complex functions optimized versions of which make it practical to offload many applications.

7 Related Work

Networking use cases & measurements of eBPF. Significant attention has been paid to using eBPF to improve in-kernel packet processing performance, leading to the development of high-performance load balancers [65, 79], key-value stores [8, 25, 44], distributed protocols [83, 84] and DDoS mitigation engines [11, 52]. Other works have customized the kernel to reduce network overheads by developing adaptive TCP/IP stack enhancements [5], efficient traffic filtering [49], fine-grained congestion control decisions [29], and enabling event-driven, shared-memory serverless computing [59]. Others [6, 10, 50, 53, 63] have noted the performance trade-offs inherent in these.

eBPF runtime features. A comprehensive benchmark and analysis of eBPF map types, performance characteristics, and overhead factors is presented in [46]. Challenges in implementing fast data structures and optimizing memory accesses in eBPF are explored in [51], while the overhead of the *tail-call* mechanism, a common feature in modular eBPF programs, is evaluated in [16, 37]. The safety of the eBPF JIT compiler, a critical runtime component, is analyzed in [56], which applies formal methods to verify its correctness and uncover vulnerabilities. Furthermore, empirical

studies on the development process of eBPF programs [20] highlight shortcomings in tooling and debugging resources, emphasizing the need for enhancements to better support developers.

Performance optimizations for eBPF. Efforts to optimize the performance of eBPF programs encompass both compile-time and runtime techniques. Compile-time approaches, such as [48, 76], improve code generation to reduce program size and latency, while runtime techniques, like [54], dynamically adapt programs to workload patterns. Additionally, innovative in-kernel libraries [77] enhance efficiency by providing optimized data structures and algorithms. Finally, kernel-level extensions have also been proposed to enable customizable optimizations, such as memory prefetching, to reduce overhead and improve performance across diverse workloads [15].

eBPF verifier studies. The eBPF verifier, that ensures program safety and correctness, has been extensively studied [36]. Existing studies include formal methods to verify range analysis [74] and techniques leveraging state embedding to detect logic bugs [69]. Dynamic testing approaches have also been employed to evaluate the verifier's robustness [32, 34, 35, 55, 75]. Other works combine memory sanitation and kernel mechanisms to synthesize complex eBPF programs capable of capturing correctness bugs in the verifier's logic [70].

8 Conclusion

Our results show that eBPF cannot always improve the performance of networked applications. However, our message is not that the community should abandon eBPF and move to other approaches. Rather, our message is that eBPF is not a silver bullet: it is well suited to some types of applications (*e.g.*, network functions and caches) and ill-suited to others (*e.g.*, more general applications). Thus, we should treat it as another tool in our arsenal for designing high-performance networked applications. Furthermore, improving eBPF's utility does not just require finding ways to improve or circumvent the verifier (which many of us seem to focus on), but rather on solving deeper issues in the runtime and compiler ecosystem.

Acknowledgments

This work was partially supported by a a gift from Google, a research grant from NEC Laboratories Europe, and by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Health and Digital Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

References

[1] 2016. Mutilate: memcached load generator designed for high request rates. https://github.com/ix-project/mutilate.
[2] 2024. eBPF verifier - The Linux Kernel documentation. https://www.kernel.org/doc/html/v6.8-rc7/bpf/verifier.html.
[3] 2024. Verified Path Exploration for eBPF Static Analysis. https://www.cs.rutgers.edu/news-events/news/news-item/professors-santosh-nagarakatte-and-srinivas-narayana-awarded-an-ebpf-foundation-grant.
[4] 2025. DPDK: Data Plane Development Kit. https://www.dpdk.org/about/.
[5] Sepehr Abbasi Zadeh, Ali Munir, Mahmoud Mohamed Bahnasy, Shiva Ketabi, and Yashar Ganjali. 2023. On Augmenting TCP/IP Stack via eBPF. In Workshop on EBPF and Kernel Extensions. ACM.
[6] Marcelo Abranches, Erika Hunhoff, Rohan Eswara, Oliver Michel, and Eric Keller. 2024. LinuxFP: Transparently Accelerating Linux Networking. In International Conference on Distributed Computing Systems (ICDCS). IEEE.
[7] Yaniv Agman, Jakub Sitnicki, Yonghong Song, Martin KaFai Lau, and Alexei Starovoitov. 2023. BPF Mailing List: Are BPF programs preemptible? https://lore.kernel.org/bpf/878rhty100.fsf@cloudflare.com/t/.
[8] Zaafar Ahmed, Muhammad Hamad Alizai, and Affan A. Syed. 2018. InKeV: in-kernel distributed network virtualization for DCN. In SIGCOMM Computer Communication Review (CCR), Volume: 46, Issue: 3. ACM.
[9] Tom Barbette, Georgios P. Katsikas, Gerald Q. Maguire, and Dejan Kostić. 2019. RSS++: load and state-aware receive side scaling. In International Conference on Emerging Networking EXperiments and Technologies (CoNEXT). ACM.

[10] Theophilus A. Benson, Prashanth Kannan, Prankur Gupta, Balasubramanian Madhavan, Kumar Saurabh Arora, Jie Meng, Martin Lau, Abhishek Dhamija, Rajiv Krishnamurthy, Srikanth Sundaresan, Neil Spring, and Ying Zhang. 2024. NetEdit: An Orchestration Platform for eBPF Network Functions at Scale. In Special Interest Group on Data Communication (SIGCOMM). ACM.
[11] Gilberto Bertin. 2017. XDP in practice: integrating XDP into our DDoS mitigation pipeline. In Technical Conference on Linux Networking (Netdev).
[12] Sanjit Bhat and Hovav Shacham. 2022. Formal verification of the linux kernel eBPF verifier range analysis. Technical Report. https://sanjit-bhat.github.io/assets/pdf/ebpf-verifier-range-analysis22.pdf
[13] Matthew Butrovich, Samuel Arch, Wan Shen Lim, William Zhang, Jignesh M Patel, and Andrew Pavlo. 2025. BPF-DB: A Kernel-Embedded Transactional Database Management System For eBPF Applications. In Proceedings of the ACM on Management of Data (PACMMOD), Volume: 3, Issue: 3. ACM.
[14] Matthew Butrovich, Karthik Ramanathan, John Rollinson, Wan Shen Lim, William Zhang, Justine Sherry, and Andrew Pavlo. 2023. Tigger: A Database Proxy That Bounces with User-Bypass. In Conference on Very Large Data Bases (VLDB), Volume: 16, Issue: 11. VLDB Endowment.
[15] Xuechun Cao, Shaurya Patel, Soo Yee Lim, Xueyuan Han, and Thomas Pasquier. 2024. FetchBPF: Customizable Prefetching Policies in Linux with eBPF. In Annual Technical Conference (ATC). USENIX.
[16] Paul Chaignon. 2021. The Cost of BPF Tail Calls. https://pchaigno.github.io/ebpf/2021/03/22/cost-bpf-tail-calls.html.
[17] Paul Chaignon. 2023. eBPF Instruction Set Extensions. https://pchaigno.github.io/bpf/2021/10/20/ebpf-instruction-sets.html.
[18] Zhongjie Chen, Qingkai Meng, ChonLam Lao, Yifan Liu, Fengyuan Ren, Minlan Yu, and Yang Zhou. 2025. eTran: Extensible Kernel Transport with eBPF. In Symposium on Networked Systems Design and Implementation (NSDI). USENIX.
[19] Jonathan Corbet. 2021. A different approach to BPF loops. https://lwn.net/Articles/877062/.
[20] Mugdha Deokar, Jingyang Men, Lucas Castanheira, Ayush Bhardwaj, and Theophilus A. Benson. 2024. An Empirical Study on the Challenges of eBPF Application Development. In Workshop on EBPF and Kernel Extensions. ACM.
[21] Eric Dumazet. 2013. BPF JIT compiler. https://elixir.bootlin.com/linux/v6.11.6/source/arch/x86/net/bpf_jit_comp.c.
[22] Kumar Kartikeya Dwivedi, Rishabh Iyer, and Sanidhya Kashyap. 2024. Fast, Flexible, and Practical Kernel Extensions. In Symposium on Operating Systems Principles (SOSP). ACM.
[23] Bolaji Gbadamosi, Luigi Leonardi, Tobias Pulls, Toke Høiland-Jørgensen, Simone Ferlin-Reiter, Simo Sorce, and Anna Brunström. 2024. The eBPF Runtime in the Linux Kernel. arXiv:2410.00026 [cs.OS]
[24] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. 2019. Simple and precise static analysis of untrusted Linux kernel extensions. In Conference on Programming Language Design and Implementation (PLDI). ACM.
[25] 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 Symposium on Networked Systems Design and Implementation (NSDI). USENIX.
[26] Red Hat. 2024. Tuning the network performance. https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/9/html/monitoring_and_managing_system_status_and_performance/tuning-the-network-performance_monitoring-and-managing-system-status-and-performance.
[27] Tejun Heo. 2024. sched: Implement BPF extensible scheduler class. https://lwn.net/Articles/972075/.
[28] Tom Herbert and Willem de Bruijn. 2024. Scaling in the Linux Networking Stack. https://docs.kernel.org/networking/scaling.html.
[29] Jörn-Thorben Hinz, Vamsi Addanki, Csaba Györgyi, Theo Jepsen, and Stefan Schmid. 2023. TCP's Third Eye: Leveraging eBPF for Telemetry-Powered Congestion Control. In Workshop on EBPF and Kernel Extensions. ACM.
[30] Oliver Hohlfeld, Johannes Krude, Jens Helge Reelfs, Jan Rüth, and Klaus Wehrle. 2019. Demystifying the Performance of XDP BPF. In Conference on Network Softwarization (NetSoft). IEEE.
[31] 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 International Conference on Emerging Networking EXperiments and Technologies (CoNEXT). ACM.
[32] Hsin-Wei Hung and Ardalan Amiri Sani. 2024. BRF: Fuzzing the eBPF Runtime. In Proceedings of the ACM on Software Engineering, Volume: 1. ACM.
[33] Intel. 2024. Optimization Reference Manual: Volume 1. https://www.intel.com/content/www/us/en/content-details/671488/intel-64-and-ia-32-architectures-optimization-reference-manual-volume-1.html.
[34] IOVisor. 2018. bpf-fuzzer: Fuzzing Framework Based on libfuzzer and Clang Sanitizer. https://github.com/iovisor/bpf-fuzzer.
[35] Juan José López Jaimez and Meador Inge. 2020. Buzzer - An eBPF Fuzzer toolchain. https://github.com/google/buzzer.

[36] Guang Jin, Jason Li, and Greg Briskin. 2024. Research Report: Enhanced eBPF Verification and eBPF-based Runtime Safety Protection. In Security and Privacy Workshops (SPW). IEEE.
[37] Clement Joly and François Serman. 2020. Evaluation of tail call costs in eBPF. In Linux Plumbers Conference.
[38] Anuj Kalia, Nikita Lazarev, Leyang Xue, Xenofon Foukas, Bozidar Radunovic, and Francis Y Yan. 2025. Towards Energy Efficient 5G vRAN Servers. In Symposium on Networked Systems Design and Implementation (NSDI). USENIX.
[39] Magnus Karlsson and Björn Töpel. 2018. The path to DPDK speeds for AF XDP. In Linux Plumbers Conference.
[40] BPF Kernel Functions (kfuncs). 2024. https://docs.kernel.org/bpf/kfuncs.html.
[41] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In Symposium on Security and Privacy (SP). IEEE.
[42] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. 2000. The click modular router. In Transaction on Compuuter Systems (TOCS), Volume: 18, Issue: 3. ACM.
[43] Joanne Koong. 2022. BPF: Dynamic pointers. https://lwn.net/Articles/895885/.
[44] Kahina Lazri, Antoine Blin, Julien Sopena, and Gilles Muller. 2019. Toward an in-Kernel High Performance Key-Value Store Implementation. In Symposium on Reliable Distributed Systems (SRDS). IEEE.
[45] Tamás Lévai, Balázs Edvárd Kreith, and Gábor Rétvári. 2023. Supercharge WebRTC: Accelerate TURN Services with eBPF/XDP. In Workshop on EBPF and Kernel Extensions. ACM.
[46] Chang Liu, Byungchul Tak, and Long Wang. 2024. Understanding Performance of eBPF Maps. In Workshop on EBPF and Kernel Extensions. ACM.
[47] Toshiaki Makita, William Tu, and NV NSBU. 2020. Faster OVS Datapath with XDP. In Netdev Conference.
[48] Jinsong Mao, Hailun Ding, Juan Zhai, and Shiqing Ma. 2024. Merlin: Multi-tier Optimization of eBPF Code for Performance and Compactness. In International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). ACM.
[49] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Mauricio Vásquez Bernal, Yunsong Lu, and Jianwen Pi. 2019. Securing Linux with a faster and scalable iptables. In SIGCOMM Computer Communication Review (CCR), Volume: 49, Issue: 3. ACM.
[50] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Massimo Tumolo, and Mauricio Vásquez Bernal. 2018. Creating Complex Network Services with eBPF: Experience and Lessons Learned. In International Conference on High Performance Switching and Routing (HPSR). IEEE.
[51] Sebastiano Miano, Xiaoqi Chen, Ran Ben Basat, and Gianni Antichi. 2023. Fast In-kernel Traffic Sketching in eBPF. In SIGCOMM Computer Communication Review (CCR), Volume: 53, Issue: 1. ACM.
[52] Sebastiano Miano, Roberto Doriguzzi-Corin, Fulvio Risso, Domenico Siracusa, and Raffaele Sommese. 2019. Introducing SmartNICs in Server-Based Data Plane Processing: The DDoS Mitigation Use Case. In IEEE Access, Volume: 7. IEEE.
[53] 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. In Transactions on Network and Service Management, Volume: 18, Issue: 1. IEEE.
[54] Sebastiano Miano, Alireza Sanaee, Fulvio Risso, Gábor Rétvári, and Gianni Antichi. 2022. Domain specific run time optimization for software data planes. In International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). ACM.
[55] Mohamed Husain Noor Mohamed, Xiaoguang Wang, and Binoy Ravindran. 2023. Understanding the Security of Linux eBPF Subsystem. In SIGOPS Asia-Pacific Workshop on Systems (APSys). ACM.
[56] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. 2020. Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the Linux kernel. In Symposium on Operating Systems Design and Implementation (OSDI). USENIX.
[57] Sujin Park, Diyu Zhou, Yuchen Qian, Irina Calciu, Taesoo Kim, and Sanidhya Kashyap. 2022. Application-Informed Kernel Synchronization Primitives. In Symposium on Operating Systems Design and Implementation (OSDI). USENIX.
[58] Federico Parola, Roberto Procopio, Roberto Querio, and Fulvio Risso. 2023. Comparing User Space and In-Kernel Packet Processing for Edge Data Centers. In SIGCOMM Computer Communication Review (CCR), Volume: 53, Issue: 1. ACM.
[59] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and K. K. Ramakrishnan. 2022. SPRIGHT: extracting the server from serverless computing! high-performance eBPF-based event-driven, shared-memory processing. In Special Interest Group on Data Communication (SIGCOMM). ACM.
[60] Liz Rice. 2021. Cilium: How eBPF Streamlines the Service Mesh. https://thenewstack.io/how-ebpf-streamlines-the-service-mesh/.
[61] Benjamin Rothenberger. 2022. https://rothenberger.io/post/rss_primer/.
[62] Raj Sahu and Dan Williams. 2023. Enabling BPF Runtime policies for better BPF management. In Workshop on EBPF and Kernel Extensions. ACM.

[63] Dominik Scholz, Daniel Raumer, Paul Emmerich, Alexander Kurtz, Krzysztof Lesiak, and Georg Carle. 2018. Performance Implications of Packet Filtering with Linux eBPF. In International Teletraffic Congress (ITC). IEEE.
[64] Zi Shen Lim. 2016. BPF JIT compiler for ARM64. https://elixir.bootlin.com/linux/v6.11.6/source/arch/arm64/net/bpf_jit_comp.c.
[65] Nikita Shirokov and Ranjeeth Dasineni. 2018. Katran, a scalable network load balancer. https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer/.
[66] Jakub Sitnicki. 2022. Assembly within! BPF tail calls on x86 and ARM. https://blog.cloudflare.com/assembly-within-bpf-tail-calls-on-x86-and-arm/.
[67] Livio Soares and Michael Stumm. 2010. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In Symposium on Operating Systems Design and Implementation (OSDI). USENIX.
[68] Alexei Starovoitov. 2020. bpf: Introduce global functions. https://lore.kernel.org/bpf/20200109063745.3154913-6-ast@kernel.org/t/.
[69] Hao Sun and Zhendong Su. 2024. Validating the eBPF Verifier via State Embedding. In Symposium on Operating Systems Design and Implementation (OSDI). USENIX.
[70] Hao Sun, Yiru Xu, Jianzhong Liu, Yuheng Shen, Nan Guan, and Yu Jiang. 2024. Finding Correctness Bugs in eBPF Verifier with Structured and Sanitized Program. In European Conference on Computer Systems (EuroSys). ACM.
[71] Dave Thaler. 2024. RFC 9669: BPF Instruction Set Architecture (ISA). https://www.rfc-editor.org/rfc/rfc9669.html.
[72] Nguyen Van Tu, Jonghwan Hyun, Ga Yeon Kim, Jae-Hyoung Yoo, and James Won-Ki Hong. 2018. INTCollector: A High-performance Collector for In-band Network Telemetry. In International Conference on Network and Service Management (CNSM).
[73] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. 2021. revisiting the open vSwitch dataplane ten years later. In Special Interest Group on Data Communication (SIGCOMM). ACM.
[74] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. 2023. Verifying the Verifier: eBPF Range Analysis Verification. In Computer Aided Verification (CAV). Springer-Verlag.
[75] Dmitry Vyukov and Andrey Konovalov. 2015. Syzkaller: an unsupervised coverage-guided kernel fuzzer. https://github.com/google/syzkaller.
[76] Qiongwen Xu, Michael D. Wong, Tanvi Wagle, Srinivas Narayana, and Anirudh Sivaraman. 2021. Synthesizing safe and efficient kernel extensions for packet processing. In Special Interest Group on Data Communication (SIGCOMM). ACM.
[77] Bin Yang, Dian Shen, Junxue Zhang, Hanlin Yang, Lunqi Zhao, Beilun Wang, Guyue Liu, and Kai Chen. 2025. eNetSTL: Towards an In-kernel Library for High-Performance eBPF-based Network Functions. In European Conference on Computer Systems (EuroSys). ACM.
[78] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In Symposium on Operating Systems Design and Implementation (OSDI). USENIX.
[79] Rui Yang and Marios Kogias. 2023. HEELS: A Host-Enabled eBPF-Based Load Balancing Scheme. In Workshop on EBPF and Kernel Extensions. ACM.
[80] 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 Annual Technical Conference (ATC). USENIX.
[81] Yusheng Zheng, Yiwei Yang, Haoqin Tu, and Yuxi Huang. 2024. Code-Survey: An LLM-Driven Methodology for Analyzing Large-Scale Codebases. arXiv:2410.01837 [cs.SE]
[82] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. 2022. XRP: In-Kernel Storage Functions with eBPF. In Symposium on Operating Systems Design and Implementation (OSDI). USENIX.
[83] Yang Zhou, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. 2023. Electrode: Accelerating Distributed Protocols with eBPF. In Symposium on Networked Systems Design and Implementation (NSDI). USENIX.
[84] Yang Zhou, Xingyu Xiang, Matthew Kiley, Sowmya Dharanipragada, and Minlan Yu. 2024. DINT: Fast In-Kernel Distributed Transactions with eBPF. In Symposium on Networked Systems Design and Implementation (NSDI). USENIX.

Received December 2024; revised June 2025; accepted June 2025