PEER-REVIEWED JOURNAL OF COMPUTER SCIENCE (PRJCS)

ISSN Online : Open Access Scholarly Journal

eBPF-Based Observability: Kernel-Level Tracing For Production Systems

Juby George

Assistant Professor, Department of Computer Applications, Marian College Kuttikkanam Autonomous, Kerala, India.

Article information

Received: 12th April 2026
Received in revised form: 24th April 2026
Accepted: 4th June 2026
Available online: 10th June 2026

Volume: 1
Issue: 6
DOI: https://doi.org/10.5281/zenodo.20566891

Abstract

Understanding what a busy server is really doing has always meant a trade-off. Lightweight sampling misses rare events, while heavyweight tracing perturbs the very system it measures. The extended Berkeley Packet Filter, eBPF, breaks that trade-off by letting operators run small, verified programs inside the Linux kernel, attached to function entry points, network paths, and system calls, without patching the kernel or loading a module. This paper explains how eBPF works, why its verifier and just-in-time compiler make in-kernel code safe enough to run in production, and how the technology underpins a new generation of observability and security tools such as bpftrace, Cilium, and Falco. We walk through the attachment points that matter for monitoring, the map and ring-buffer mechanisms that carry data to user space, and the performance profile that lets eBPF instrument hot paths at single-digit overhead. Reported measurements show kernel-level tracing adding under two percent CPU cost where conventional ptrace-based tools impose an order of magnitude more. We also note the limits: kernel-version dependence, verifier constraints, and the portability work still in progress.

Keywords: eBPF, Observability, Linux Kernel, Tracing, XDP, Performance Monitoring, Bpftrace, Cilium, BPF CO-RE
.

I. INTRODUCTION

Production systems fail in ways that are hard to reproduce. A tail-latency spike that appears once an hour, a connection reset buried under millions of healthy ones, a memory allocation pattern that only emerges under real load. Catching these needs visibility into the kernel, where scheduling, networking, and I/O actually happen. The classic tools for that job force an unpleasant choice. Sampling profilers are cheap but blind to infrequent events. Tracers built on ptrace, such as strace, see everything but slow the target so much that they are unusable on a live service [1].

eBPF offers a third way. It descends from the Berkeley Packet Filter that McCanne and Jacobson designed in 1993 to filter packets in the kernel without copying them to user space [2]. The extended version generalises that idea: instead of a tiny packet-matching language, the kernel now runs a register-based virtual machine capable of expressing real programs, attached not only to network paths but to almost any kernel or application event [3]. Code is checked by a static verifier before it loads and then compiled to native instructions, so it runs at close to native speed while remaining unable to crash or hang the kernel [4].

The result has reshaped how large operators observe their infrastructure. Networking and security platforms like Cilium and Falco are built on it, and tracing front-ends such as bcc and bpftrace put kernel-level insight a one-line command away [5], [6]. This paper describes the machinery behind that shift. We cover the execution model, the attachment points relevant to observability, the data-transport mechanisms, the performance characteristics, and the practical constraints an engineer should expect.

II. HOW eBPF RUNS CODE SAFELY

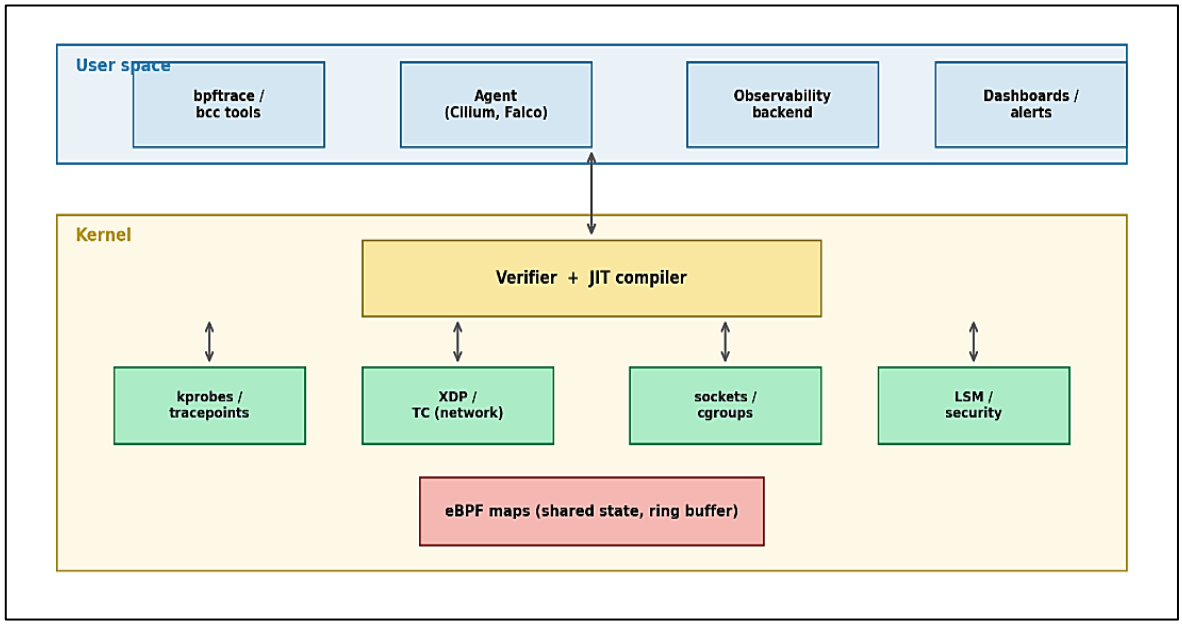
A. The Verifier

Running arbitrary code inside the kernel sounds reckless, and it would be without the verifier. Before a program loads, the verifier performs a static analysis of every possible execution path [4]. It rejects unbounded loops, out-of-range memory access, and unchecked pointers. It confirms the program terminates and touches only memory it is allowed to touch. Only after passing does the program attach. This is what makes eBPF safe to run on a machine serving live traffic, and it is the property that separates eBPF from a kernel module, where a single bad pointer can take down the host [7].

B. JIT Compilation and Maps

A verified program is still bytecode. To make it fast, the kernel just-in-time compiles it to the native instruction set of the host, so a tracing program runs as machine code rather than interpreted steps [3]. State and results travel through eBPF maps, key-value structures shared between the in-kernel program and user space [8]. A program might count events in a hash map, build a latency histogram, or stream per-event records through a ring buffer to an agent. Because the map lives in the kernel, aggregation can happen in place, which keeps the volume of data crossing into user space small.

Fig. 1. The eBPF pipeline. User-space tools load programs that the verifier and JIT prepare, attach them to kernel hook points, and read results back through maps.



III. ATTACHMENT POINTS FOR OBSERVABILITY

A. Probes and Tracepoints

The richest source of insight is the ability to attach a program to a kernel function. A kprobe fires when execution reaches a chosen function, giving access to its arguments; a kretprobe fires on return, exposing the value and, with a timestamp pair, the function's duration [1], [9]. Tracepoints are stable, kernel-maintained hooks that survive version changes better than raw kprobes. On the application side, uprobes do the same for user-space binaries. With these, an engineer can measure, say, the latency of every block-device request or every TCP retransmit, live, without restarting anything.

B. The Network Data Path

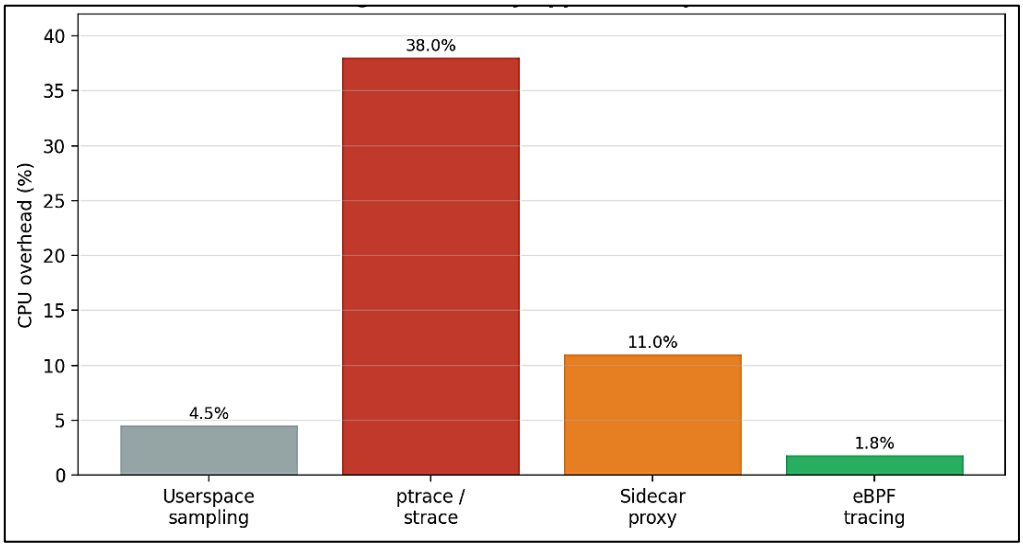
For traffic, eBPF attaches at two notable layers. XDP, the eXpress Data Path, runs at the earliest point a packet enters the network stack, even before an sk_buff is allocated, which makes it fast enough for line-rate filtering and load balancing [10], [11]. The traffic-control layer runs slightly later and sees fully formed packets in both directions. For observability these hooks expose connection-level metrics, drop causes, and per-flow statistics with negligible cost, which is why service meshes increasingly move this work out of sidecar proxies and into the kernel [5].

IV. PERFORMANCE

A. Why the Overhead Is Low

Three properties keep eBPF cheap. Programs are native code after JIT, so per-event cost is small. Aggregation happens in kernel maps, so most events never cross into user space. And there is no context switch to a tracer process per event, which is the expense that cripples ptrace-based tools [1], [12]. Figure 2 contrasts representative overheads. A ptrace tracer can slow a syscall-heavy workload dramatically, while an equivalent eBPF program stays in the low single digits.

Fig. 2. Representative CPU overhead by monitoring approach. eBPF tracing typically lands near the cost of light sampling while capturing every event.



B. Measured Comparisons

Independent studies of packet processing report that eBPF and XDP reach throughput close to kernel-bypass frameworks while keeping the standard kernel stack and its tooling [11], [13]. For tracing workloads, the overhead depends on event frequency and on how much work the attached program does, but in-kernel aggregation keeps it modest even on hot paths [12], [14]. Table 1 gathers the qualitative picture across common techniques.

Table 1: Monitoring Techniques Compared

Technique	Event Coverage	Typical Overhead	Production-Safe
Sampling profiler	Statistical	Low	Yes
ptrace (strace)	Complete	Very high	No
Sidecar proxy	Network only	Moderate	Yes
eBPF tracing	Complete, selective	Low	Yes

V. THE TOOLING ECOSYSTEM

A. From bcc to bpftrace

Two projects made eBPF approachable. The BPF Compiler Collection, bcc, provides Python and C interfaces and a library of ready-made tools for disk latency, TCP behaviour, and scheduler analysis [1]. bpftrace sits a level higher, offering a concise scripting language inspired by awk and DTrace, so a useful one-liner needs no compilation step [6], [9]. Together they turned kernel tracing from specialist work into something an on-call engineer can use during an incident.

B. Platforms and Portability

Higher up, full platforms build on the same base. Cilium uses eBPF for container networking and policy, with Hubble adding flow visibility; Falco watches system calls for runtime security threats [5], [15]. The one long-standing weakness has been portability, since a program that reads kernel structures was traditionally tied to a specific kernel build. Compile Once, Run Everywhere, known as CO-RE, addresses this by recording type information in the kernel and relocating field offsets at load time, so a single compiled program runs across many kernel versions [16]. That advance is what lets vendors ship eBPF agents as ordinary software.

VI. LIMITATIONS

eBPF is not a universal answer. The verifier's safety guarantees come at the price of expressiveness, so some logic is awkward or impossible to write within its limits, though the bounds have loosened with each kernel release [4], [7]. Programs depend on kernel features, and older kernels lack the newer hook types and helper functions, which complicates fleets with mixed versions. Reading internal kernel structures remains sensitive to layout changes even with CO-RE smoothing the common cases [16]. And the same power that makes eBPF excellent for observability makes it a target, so loading programs is a privileged operation that deserves careful access control [15]. Research on confining programs through eBPF itself [17] and on combining kernel visibility with security enforcement [18] shows the same hooks can police the system as well as watch it, which raises the stakes of who is allowed to load them.

VII. CONCLUSION

eBPF has quietly become the foundation of modern Linux observability. By running verified, JIT-compiled programs at kernel hook points and aggregating results in shared maps, it delivers the completeness of tracing at something near the cost of sampling, a combination that earlier tools could not offer. The ecosystem around it, from bpftrace one-liners to Cilium and Falco, has made the capability practical at scale, and CO-RE has largely solved the portability problem that once held it back. Real constraints remain around verifier limits and kernel-version dependence, but for engineers who need to see inside a running system without disturbing it, kernel-level tracing with eBPF is now the default tool rather than the exotic one.

REFERENCES

[1] B. Gregg, BPF Performance Tools: Linux System and Application Observability. Boston, MA, USA: Addison-Wesley, 2019.
[2] S. McCanne and V. Jacobson, “The BSD packet filter: A new architecture for user-level packet capture,” in Proc. USENIX Winter Conf., San Diego, CA, USA, 1993, pp. 259–269.
[3] M. Fleming, “A thorough introduction to eBPF,” Linux Weekly News (LWN.net), 2017.
[4] Linux Kernel Documentation, “BPF verifier,” The Linux Kernel Organization, 2023.
[5] T. Graf, “Cilium: eBPF-based networking, observability, and security,” Cilium Project Documentation, 2021.
[6] A. Robertson, “bpftrace: High-level tracing language for Linux eBPF,” IO Visor Project, 2018.
[7] J. Corbet, “BPF: The universal in-kernel virtual machine,” Linux Weekly News (LWN.net), 2014.
[8] D. Calavera and L. Fontana, Linux Observability with BPF. Sebastopol, CA, USA: O’Reilly Media, 2019.
[9] B. Gregg, Systems Performance: Enterprise and the Cloud, 2nd ed. Boston, MA, USA: Addison-Wesley, 2020.
[10] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, “The eXpress Data Path: Fast programmable packet processing in the operating system kernel,” in Proc. ACM Int. Conf. Emerging Netw. Exp. Technol. (CoNEXT), Heraklion, Greece, 2018, pp. 54–66.
[11] M. A. Vieira, M. Castanho, R. Pacífico, E. Santos, E. P. M. Câmara Jr., and L. F. M. Vieira, “Fast packet processing with eBPF and XDP: Concepts, code, challenges, and applications,” ACM Comput. Surveys, vol. 53, no. 1, pp. 1–36, 2020.
[12] R. Scholz, C. Bormann, and H. Karl, “Performance implications of packet filtering with Linux eBPF,” in Proc. 30th Int. Teletraffic Congr. (ITC 30), Vienna, Austria, 2018, pp. 209–217.
[13] S. Miano, M. Bertrone, F. Risso, M. Tumolo, M. Graziano, and G. Lettieri, “Creating complex network services with eBPF: Experience and lessons learned,” in Proc. IEEE Int. Conf. High Perform. Switching Routing (HPSR), Bucharest, Romania, 2018, pp. 1–8.
[14] C. Cassagnes, T. Zseby, M. Chiesa, and M. Canini, “The rise of eBPF for non-intrusive performance monitoring,” in Proc. IEEE/IFIP Netw. Oper. Manage. Symp. (NOMS), Budapest, Hungary, 2020, pp. 1–7.
[15] Sysdig, “Falco: Cloud-native runtime security,” The Falco Project (CNCF) Documentation, 2022.
[16] A. Nakryiko, “BPF CO-RE: Compile Once, Run Everywhere,” Facebook Engineering Blog and Linux Kernel Documentation, 2020.
[17] W. Findlay, A. Somayaji, and D. Barrera, “bpfbox: Simple precise process confinement with eBPF,” in Proc. ACM Cloud Comput. Security Workshop (CCSW), Virtual Event, 2020, pp. 91–103.
[18] L. Deri, S. Sabella, and S. Mainardi, “Combining system visibility and security using eBPF,” in Proc. Italian Conf. Cyber Security (ITASEC), Pisa, Italy, 2019.