Survey of eBPF: Applications and Vulnerabilities

Carol Hsu, Weiser Chen
hsuc14@uci.edu,yingchc6@uci.edu

ABSTRACT

This paper investigates the application of eBPF in enhancing transaction management, particularly through its integration in networking and observability frameworks. We analyze the capabilities of several open-source tools—including Cilium[11] and Falco[8] — that utilize eBPF to monitor and enforce system behavior. Additionally, we review recent research, with a focus on network and security in eBPF. Our investigation demonstrates the trend in eBPF-based solutions, and these solutions can significantly improve the security, efficiency and adaptability for modern complicated network environment.

1 INTRODUCTION

eBPF has gained popularity in recent years as a high-performance approach for networking, observability, and security. eBPF programs are loaded into the kernel and attached to tracepoints, which are triggered by events such as system calls and network operations. The technology is leveraged by many open-source projects to enhance performance and functionality.

2 BACKGROUND

eBPF is a modern mechanism that allows developers to modify kernel behavior without altering the kernel code directly, offering significant flexibility to run programs in kernel space securely. Its introduction has unlocked numerous possibilities by enabling new execution paths within the kernel, creating both opportunities and potential vulnerabilities. For instance, many networking infrastructures leverage eBPF to bypass parts of the traditional network stack to achieve better performance. Unlike kernel modules, eBPF programs must first pass a strict verifier before being loaded into the kernel, greatly reducing the risk of kernel crashes. However, this also introduces a new attack surface, providing a more obscure avenue for exploitation if root privileges are compromised. This section summarize materials from [7] and [17].

2.1 eBPF programs

An eBPF system consists of two components: the eBPF program itself and a user-space program. The user-space program is responsible for loading the eBPF program into the kernel. To facilitate communication between the two, maps, a type of global data structure, are used. Each eBPF program can only run in a single-threaded context, but it can access maps shared by other programs. To inspect and manage eBPF programs, tools like **bpftool**[3] are commonly used. Once loaded, an eBPF program must be attached to specific kernel events to execute. Common attachment points include system calls, tracepoints, and kprobes. More details are provided in the following sections. eBPF maps can take various forms, such as hash maps, perf buffers, and ring buffers. Due to stack size limitations in eBPF, programs can invoke other eBPF functions using tail calls. To do this, the callee programs must be stored in an array map, and the **bpf_tail_call()** helper function is used to transfer execution context to the target program.

eBPF programs interact with the system through a set of syscalls, typically prefixed with **bpf***. The kernel maintains references to eBPF programs and maps, and automatically releases them when they are no longer referenced. To prevent premature release, bpftool can be used to pin programs or maps to a path in the file system, thereby maintaining their references. When monitoring events, **epoll** is generally preferred over ppoll, as ppoll requires re-registering events after each return. In contrast, epoll retains the event registration within the kernel until explicitly removed, making it more efficient for persistent event handling.

2.2 CO-RE

To improve the portability of eBPF programs across different kernel versions, CO-RE (Compile Once, Run Everywhere) was introduced in newer versions of the Linux kernel. Unlike BCC, a widely used experimental framework for eBPF that compiles code dynamically, CO-RE allows for generating relocatable bytecode that can be adjusted at load time. This approach enhances stability and reusability. BTF (BPF Type Format) encodes type information used for debugging and for adapting the program to the kernel's data structures during loading. The file **vmlinux.h** contains all kernel data structures in BTF format and is typically included in eBPF programs. Generating relocation code requires compiler support, but this is now provided by Clang, GCC, and Rust. Several libraries are available to handle the loading process and automatically perform the necessary relocations of the bytecode.

2.3 eBPF verifier

The eBPF verifier checks the program before it is loaded into the kernel. It analyzes all possible execution paths and enforces a strict limit: the total number of instructions executed across all paths must not exceed one million. Because the program may contain branches and loops, the effective instruction limit per path is often much lower. When the verifier encounters a branch, it saves the current state as a checkpoint. After completing one path, it restores the previous state from the stack to explore alternative paths. To ensure memory safety, the verifier checks that all memory accesses stay within valid bounds. It tracks the value ranges of registers and raises an error if a value is determined to be out of bounds. Each BPF helper function is associated with a prototype that specifies the function and its constraints, allowing the verifier to enforce proper usage. The verifier also checks that all pointers are validated against null before being dereferenced. Access to certain fields in the context structure is restricted and not allowed. Initially, loops were not permitted in eBPF unless they were fully unrolled. However, recent kernel versions allow loops, and using the **bpf_loop()** helper can simplify the verification process. Additionally, the program must define a valid return code. All eBPF instructions must be syntactically correct, and unreachable instructions are not permitted. Some

BPF helper functions also require the program to be released under the GPL license.

2.4 Tracing

eBPF program types can only be attached to their corresponding attachment points. For example, retrieving the user ID in an XDP (eXpress Data Path)[21] program is not meaningful, since XDP operates at the network layer and has no concept of user identity, only raw network packets. Each program type receives a different context structure as input, and the set of available helper functions also varies depending on the type. As mentioned earlier, only certain helper functions are permitted for specific program types. The return value of an eBPF program can also influence the behavior of normal kernel processing, such as dropping or redirecting a packet.

Tracepoints and raw tracepoints are stable trace locations exposed by the kernel, often used for observability. Kprobes and kretprobes attach to the entry and return of kernel functions. Fentry and fexit are similar in purpose to kprobes, but fexit additionally allows access to input arguments when a function returns. Uprobes and uretprobes are designed to attach to user-space code, such as functions in shared libraries. However, they may not be invoked in some cases, such as when dealing with statically linked binaries, containerized file systems, architecture- or distribution-specific paths, or programming runtimes like Go or Java. Another important attachment type is LSM (Linux Security Module), which hooks into security-related events within the kernel.

In the networking stack, eBPF can interact with several subsystems, including sockets, traffic control (tc), XDP, and cgroups, enabling flexible packet processing and network policy enforcement.

2.5 Programming Framework

bpftrac[4] is a tool that allows developers to write eBPF programs without needing to manage the division between user space and kernel space. It uses a high-level scripting language to describe eBPF behavior and is particularly well-suited for simple tracing tasks. It is often used for quick diagnostics and has an accessible tutorial for beginners.

BCC[10] in Python dynamically compiles eBPF code at runtime. This approach requires the presence of a compilation toolchain on the target machine. While BCC is useful for experimentation and rapid development, it is generally recommended to use CO-RE in production environments. This involves compiling eBPF code into portable bytecode and loading it through dedicated user-space programs. The IOVisor project provides tutorials and examples to get started.

C with libbpf[13] is the most native and widely adopted approach for production use. Typically, only the eBPF code is written in C, while the user-space components are written in other higher-level languages. This method offers low-level control and better performance.

In Go, the cilium/ebpf[6] library provides functionality similar to libbpf and supports skeleton generation to manage the lifecycle of eBPF objects. Another project, libbpfgo[18], offers Go bindings for libbpf and allows integration with native Go features such as channels. However, it can introduce complications at the C/Go boundary, such as memory safety concerns or interface mismatches.

In Rust, Aya[1] is the most prominent eBPF framework. It supports targeting eBPF bytecode directly, allowing both the kernel-space and user-space code to be written in Rust. This unification enables reuse of data structures and type definitions across both contexts, simplifying development and improving maintainability.

3 OPEN SOURCES TOOL

3.1 Cillium

Cilium[11] is an open-source, cloud-native networking solution designed to provide, secure, and observe network connectivity between workloads using eBPF (Extended Berkeley Packet Filter). Unlike traditional networking models, Cilium leverages eBPF to enable highly efficient packet processing, observability, and security. One of the most common applications of Cilium is Kubernetes network management. While Kubernetes has built-in networking capabilities, such as network policies and pod traffic control, Cilium offers a more advanced and flexible solution. By integrating with Kubernetes, Cilium enhances service-to-service communication, security policies, and load balancing, making it a good choice for large-scale, complex deployments. Cilium actually provides L2-L4 firewall policy, traffic monitoring, and dynamic security policies. On top of Cilium, Hubble[12], provides real-time network observability and troubleshooting tools, making it easier to monitor traffic flows and security posture across distributed environments.

As of June 2025, Cilium has 21.7k stars on GitHub, reflecting its growing adoption and a strong open-source community. The project continues to evolve, with extensive documentation, active development, and contributions from cloud providers, enterprises, and individual developers. Thanks to its user-friendly introduction and large, engaged developer community, Cilium is expected to keep advancing cloud networking technologies, making eBPF-based networking a mainstream solution in the industry.

3.2 Falco

Falco[8] is an open-source project under the CNCF that detects security events by analyzing logs from various sources. The system is structured into two main components: local agents and log aggregators. Local agents are deployed on individual machines to monitor security-related activities, such as system calls. These collected logs are then forwarded to a centralized log aggregator, where they are processed and evaluated against predefined security policies.

Log sources can include files or other data streams that conform to either Falco's built-in formats or those supported through third-party extensions. Traditionally, Falco relied on a kernel module to intercept system activity. However, with the growing adoption of eBPF, the system now uses eBPF as the default logging mechanism. eBPF offers enhanced flexibility and security, allowing deeper and safer introspection of kernel events without requiring intrusive kernel modifications. By leveraging eBPF, Falco achieves more secure and efficient runtime monitoring, making it suitable for modern containerized and cloud-native environments.

4 PAPERS

4.1 eTran: Extensible Kernel Transport with eBPF

Evolving datacenter demands are reshaping network transport design, yet integrating new transport mechanisms (e.g., Homa, DCTCP, NDP) into the widely used kernel networking stack remains a slow and complex process. This paper introduces eTran[5], a system that enables agile customization of transport protocols within the kernel, making kernel transport extensible for multiple applications need.

The main foundation of eTran is XDP, XDP (eXpress Data Path)[20] is an eBPF hook integrated into the Linux kernel, designed for high-speed packet processing. AF_XDP, a specialized socket family built on XDP, enables fast packet handling in user space. Packets processed through XDP can be dropped, forwarded, redirected to AF_XDP sockets, or passed to kernel networking stacks.

Simply put, eTran processes packets through XDP, supports various transport layer protocols, and enables user-customized congestion control. Its design separates functionality into a data plane and a control plane: the data plane handles performance-critical transport operations (e.g., packet reassembly/ACK), while the control plane manages non-performance-sensitive tasks such as configuration and eBPF program management

To conclusion, eTran enables secure offloading of complete transport states and performance-critical operations into the kernel, ensuring robust protection. It facilitates key transport functionalities such as packet acknowledgment, flow pacing, and fast retransmission within eBPF. Results show TCP (with DCTCP congestion control) and Homa under eTran achieves up to 4.8×/1.8× higher throughput and 3.7×/7.5× lower latency compared to existing kernel implementations.

4.2 SeaK: Rethinking the Design of a Secure Allocator for OS Kernel

SeaK, representing Secure allocator for Kernel. This paper proposed a atomic alleviation concept to protect Linux kernel from heap attack, which is implemented by using eBPF.

Heap management in Linux is related to the slab allocator and buddy allocator. Traditionally, the allocator is a core kernel subsystem that is invoked frequently. General mindset is try to protect every object at all times, leading to performance and memory overhead with each allocation. This level of overhead is unacceptable for the kernel, as it must efficiently provide services to user space while operating with limited memory.This work introduces a novel strategy centered on the concept of atomic alleviation. Basically, atomic alleviation is all about isolating specific types of kernel objects to strengthen security. Instead of protecting every objects all the time, atomic alleviation focuses only on the most critical parts, making it more efficient. Plus, it do not need to reboot or recompile the kernel while applying.
The implementation consists of two parts:

- **Synthesis Template**
- **Determining Allocation and Free Sites**

In the **synthesis phase**, the `kpi_type` (Kernel Programming Interface, KPI) information in the eBPF program is derived from the kernel image rather than the source code to eliminate inaccuracies, ensuring compatibility with the actual kernel environment. In the **determining phase**, SeaK identifies allocation and free sites, then converts these locations into their corresponding binary addresses called Guard Page which using randomization mechanism. In brief, the eBPF program synthesizer in user space deploys an eBPF program into kernel space, incorporating allocation and free handlers to isolate specific objects of interest.

Using real-world cases for evaluation, this paper compares the ability to isolate security-sensitive objects and mitigate vulnerabilities across three competitors, without explicitly naming them. The results show that SeaK outperforms all others. In conclusion, SeaK provides a scalable and stable solution for protecting against heap attacks, demonstrating low cost and high reliability. Although the eBPF ecosystem has previously been reported to contain vulnerabilities[14], [15], it still has potential for further development, such as protecting stack attacks in the future.

4.3 Cross-Containers Attack

The Cross-Container Attack[9] paper explores how eBPF can be exploited to compromise containerized environments. The authors demonstrate several attacks that leverage malicious use of bpf system calls to perform denial-of-service (DoS), extract sensitive kernel information, and potentially break container isolation. eBPF programs are a powerful mechanism for executing code in kernel space and are widely used for tasks such as network filtering and system tracing. However, because these programs require access to kernel-level data, they typically need elevated privileges, most notably the **CAP_SYS_ADMIN** capability. If granted, this access becomes a powerful attack vector in multi-container environments where containers share the same kernel.

The attacks described in the paper begin with identifying containers that run with root privileges and allow the use of bpf syscalls. Once such a container is compromised, an attacker can load a malicious eBPF program into the kernel, enabling actions such as stealing data from other containers, terminating processes across container boundaries, or even escalating privileges to the host.

The paper includes an empirical study of DockerHub images, revealing that approximately 2.5% of all images and over 3% of the top 300 images run as root and enable the use of privileged system features. Many of these are tied to Kubernetes-based workloads that involve system monitoring or network policy enforcement, which justifies their elevated privileges. However, this also makes them attractive targets.

In Kubernetes environments, privileged components like Operators, CronJobs, and DaemonSets are responsible for background system tasks such as log aggregation, health checks, and network configuration. These components often run with elevated permissions, which can be exploited. For example, if an attacker gains root access to a pod, they may be able to impersonate its service account and propagate the attack cluster-wide by targeting other DaemonSet instances running on different nodes. This significantly expands the attack surface beyond a single container.

For responsible disclosure, the authors coordinated with major vendors to assess and test the attack feasibility. Some vendors provided dedicated test environments, while others allowed isolated testing within their platforms. The researchers reported the vulnerabilities to the affected parties and issued warnings to the Docker and Kubernetes communities. Their recommendations include disabling bpf system calls in containers that do not require them, tightening the use of privileged containers, and applying stricter admission controls to prevent unnecessary access to powerful capabilities.

In summary, the paper highlights an important yet often overlooked security risk in modern container orchestration systems: the misuse of powerful kernel interfaces like eBPF in shared environments. It advocates for more principled security boundaries and fine-grained privilege management in container deployments.

4.4 DeepFlow

DeepFlow[19] introduces a distributed tracing system with a network-centric design. Unlike traditional distributed tracing frameworks that operate mainly at the application level, DeepFlow focuses on the network layer, where a significant portion of production issues originate, such as packet loss, connection drops, and NIC malfunctions. Traditional tracing solutions often require code instrumentation, either manually through trace logging or automatically through compiler or interpreter hooks. For example, OpenTelemetry[16] supports both automated instrumentation for many languages and manual logging for finer control. While these frameworks are effective for tracking application-level issues, they generally lack visibility into low-level network events.

DeepFlow addresses this gap by leveraging eBPF to implement zero-code instrumentation at the kernel level. It captures detailed network traces without modifying application code. The tracing interface is abstracted into a well-defined ABI, divided into ingress/egress and enter/exit points. This allows DeepFlow to track interactions across microservices and reconstruct full trace spans based on observed kernel-level events. The architecture consists of two components: the Agent, which runs on each node to collect tracing data, and the Server, which aggregates, stores, and analyzes the collected traces.

Unlike frameworks that rely on explicit context propagation (e.g., passing trace IDs through headers), DeepFlow uses implicit context propagation. It correlates trace events using network-level metadata such as thread IDs and 5-tuple connection information (source/destination IPs and ports, protocol). This allows it to infer causal relationships between events without modifying application payloads. Additionally, DeepFlow supports tag-based correlation using identifiers such as Kubernetes labels, resource tags, and commit IDs. To minimize storage overhead, tags are converted into integer IDs, with a separate mapping table storing the original string-to-integer relationships.

In terms of performance, DeepFlow introduces a relatively low overhead. According to the evaluation in the paper, the system adds about 5% additional latency per request, making it practical for real-world deployment.

To conclude, DeepFlow is a compelling system for network-centric observability. Its kernel-level instrumentation enables visibility into low-level behaviors that traditional tracing frameworks cannot capture, making it particularly valuable for diagnosing networking failures and detecting security-related anomalies. By monitoring the entire communication path between services, DeepFlow can help identify the exact component responsible for an issue. Moreover, its architecture and tagging mechanism make it well-suited for large-scale systems like Kubernetes. DeepFlow could also be extended beyond tracing to support offline analysis and anomaly detection, making it a powerful tool for both real-time monitoring and forensic investigation.

4.5 NetEdit

NetEdit[2] is an eBPF orchestration platform developed by Meta to optimize network behavior in response to varying workloads. Its primary goal is to enhance performance by managing the lifecycle of eBPF programs more effectively. NetEdit introduces a unified interface that abstracts away kernel version differences and low-level implementation details, allowing operators to dynamically tune network parameters without dealing with underlying system complexity. By leveraging NetEdit, Meta reports a significant improvement in service performance, about a 3x increase, highlighting its effectiveness in production environments.

A key feature of NetEdit is its improvement of deterministic execution for eBPF programs. In the kernel's default behavior, the order in which programs execute at a given hook point can vary: some are executed in the order they were installed, while others rely on priority flags. NetEdit introduces a layer of indirection by storing all programs for a given hook point in a dedicated array. This array is organized based on predefined priorities, enabling controlled and predictable execution order. This design reduces the risk of subtle bugs that might arise when multiple eBPF programs interact or interfere with one another due to nondeterministic execution sequences.

While the paper focuses primarily on networking use cases, it briefly touches on other potential applications. One notable area is security. Although the system currently does not emphasize security functionality, the idea of eBPF orchestration could be extended to support dynamic security policy management. For example, firewall rules and access control policies could be updated in real time in response to detected threats or ongoing attacks. This real-time, policy-driven reconfiguration could enhance system resilience and response time in critical scenarios. Overall, NetEdit demonstrates that orchestrating eBPF programs at scale can bring both performance and operational benefits, and opens up avenues for future research in areas like real-time security enforcement and policy-driven systems.

5 CONCLUSION

Our survey illustrates that eBPF is a powerful tool that enables safe, high-performance execution of custom logic inside the Linux kernel—without modifying kernel code. Its ability to bypass network stack(XDP) and integrate deeply with tracing, observability, and security frameworks highlights its value in dynamic environments such as cloud-native and containerized systems.

The increasing users on eBPF does not necessarily imply flaws in the original kernel or network stack design, but rather reflects the evolving complexity and customization needs of modern infrastructure. While eBPF empowers developers with fine-grained control, the more code be writen, the more potential bugs will be introduced. Despite its convenience, practical use of eBPF can expose previously overlooked , especially in security-sensitive domains. For example, when bypassing the network stack to implement customized protocol logic (e.g., TCP reassembly), it is extremely difficult to replicate the robustness and correctness of decades-old kernel implementations without introducing flaws.

Additionally, although eBPF does not need to update kernel code, it is not entirely isolated from kernel updates. Changes in kernel object structures require updates to eBPF bytecode and metadata. Thus, developers must still track kernel changes closely and keep their eBPF programs up-to-date to ensure continued compatibility and correctness.

In summary, while eBPF opens up a powerful new paradigm for system-level customization and introspection, it also introduces new responsibilities and attack surfaces. The community must balance innovation with testing, version control, and security awareness as adoption continues to grow.

6 WORK DISTRIBUTION

- Weiser: sections 2, 3.2, 4.3, 4.4, 4.5
- Carol: sections 1, 3.1, 4.1, 4.2, 5

REFERENCES

[1] aya. 2025. aya. https://github.com/aya-rs/aya.

[2] 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 *Proceedings of the ACM SIGCOMM 2024 Conference* (Sydney, NSW, Australia) *(ACM SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 721–734. https://doi.org/10.1145/3651890.3672227

[3] bpftool. 2025. bpftool. https://bpftool.dev/.

[4] bpftrace. 2025. bpftrace. https://bpftrace.org/.

[5] Zhongjie Chen, Qingkai Meng, ChonLam Lao, Yifan Liu, Fengyuan Ren, Minlan Yu, and Yang Zhou. 2025. eTran: Extensible Kernel Transport with eBPF. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association, Philadelphia, PA, 407–425. https://www.usenix.org/conference/nsdi25/presentation/chen-zhongjie

[6] Cilium. 2025. Cilium eBPF. https://github.com/cilium/ebpf.

[7] eBPF. 2025. What is eBPF. https://ebpf.io/what-is-ebpf/.

[8] Falco. 2025. Falco. https://falco.org/.

[9] Yi He, Roland Guo, Yunlong Xing, Xijia Che, Kun Sun, Zhuotao Liu, Ke Xu, and Qi Li. 2023. Cross Container Attacks: The Bewildered eBPF on Clouds. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 5971–5988. https://www.usenix.org/conference/usenixsecurity23/presentation/he

[10] IOVisor. 2025. BCC. https://github.com/iovisor/bcc.

[11] Isovalent. 2025. Cilium: eBPF-based Networking, Security, and Observability. https://cilium.io. Accessed June 5, 2025.

[12] Isovalent. 2025. Network Observability with Hubble. https://docs.cilium.io/en/stable/observability/hubble/#hubble-intro. Accessed June 5, 2025.

[13] libbpf. 2025. libbpf. https://github.com/libbpf/libbpf.

[14] MITRE. 2022. CVE-2021-4204. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-4204 Accessed: 2025-06-07.

[15] MITRE. 2022. CVE-2022-0264. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-0264 Accessed: 2025-06-07.

[16] OpenTelemetry. 2025. OpenTelemetry. https://opentelemetry.io/.

[17] Liz Rice. 2025. Learning eBPF. https://isovalent.com/books/learning-ebpf/.

[18] Aqua Security. 2025. libbpfgo. https://github.com/aquasecurity/libbpfgo.

[19] Junxian Shen, Han Zhang, Yang Xiang, Xingang Shi, Xinrui Li, Yunxi Shen, Zijian Zhang, Yongxiang Wu, Xia Yin, Jilong Wang, Mingwei Xu, Yahui Li, Jiping Yin, Jianchang Song, Zhuofeng Li, and Runjie Nie. 2023. Network-Centric Distributed Tracing with DeepFlow: Troubleshooting Your Microservices in Zero Code. In *Proceedings of the ACM SIGCOMM 2023 Conference* (New York, NY, USA) *(ACM SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 420–437. https://doi.org/10.1145/3603269.3604823

[20] TheIOVisorProject. 2025. eXpress Data Path (XDP). https://www.iovisor.org/technology/xdp Accessed: 2025-06-06.

[21] xdp. 2025. xdp. https://prototype-kernel.readthedocs.io/en/latest/networking/XDP/index.html.