uXDP: Frictionless XDP Deployments in Userspace

Yusheng Zheng*
yunwei356@gmail.com
UC Santa Cruz
Santa Cruz, CA, USA

Panayiotis Gavriil*
pan.gav2001@gmail.com
The D. E. Shaw Group
New York, NY, USA

Marios Kogias
m.kogias@imperial.ac.uk
Imperial College London
London, UK

Abstract

Modern network function (NF) deployments face a fundamental trade-off: kernel-based extended Berkeley Packet Filter (eBPF) NFs provide safety, portability, and an extensive tooling ecosystem, but are limited in performance, while kernel-bypass frameworks deliver high throughput but lack integrated verification and ease of deployment. We present uXDP, a new runtime that unifies these worlds by running unmodified, verified XDP programs in userspace. uXDP ensures compatibility and preserves the verification-driven safety, portability, and familiar workflows of eBPF while moving execution into the userspace, enabling more aggressive optimizations and flexibility. Without recompiling eBPF code, uXDP achieves throughput gains of up to 3.3× over in-kernel execution and improves Meta's Katran load balancer performance by 40%, all while retaining the trusted eBPF development model and deployment simplicity.

CCS Concepts

• **Networks** → **Programmable networks**; • **Software and its engineering** → *System administration.*

Keywords

eBPF, XDP, DPDK, kernel bypass, network functions

1 Introduction

Network functions (NFs) are special-purpose programs in charge of packet processing performing tasks such as load balancing, firewalling, and NAT. Software NFs are widely deployed for their flexibility, composability, ease of use and performance. Depending on the NF complexity, current CPUs are able to process millions of packets per second on a single core. As a result, software NFs are the cornerstone of modern cloud and telco deployments.

Userspace packet processing frameworks, such as DPDK [8] and VPP [10], substantially improve the performance of software NFs. By depending on poll-mode network drivers, such frameworks eliminate the cost of interrupt processing. Also, userspace execution enables them to apply aggressive compiler optimizations such as vector instructions (e.g., SIMD) that are very efficient at batch processing of packets. However, developing and deploying such userspace processing pipelines can be quite challenging. From the developer's perspective, userspace packet processing requires implementing the logic to handle any packet that can arrive at the NIC, thus increasing the development and maintenance effort. This usually involves writing low-level code in languages such as C, which can be error-prone, thus jeopardizing the reliability of the pipeline. Also, given the lack of standard interfaces, software NFs follow an ad-hoc separation between the control and data planes which makes code porting and code reuse across projects cumbersome. From the deployment perspective, userspace packet processing requires dedicated hardware resources, such as an entire NIC or a virtual function [16], which are expensive to provision and might vary across hardware vendors, thus making hardware sharing and multiplexing hard or impossible.

The extended Berkeley Packet Filter (eBPF) [13] is emerging as a powerful technology for Linux that enables kernel extensibility by hooking eBPF programs at various points inside the kernel that are guaranteed not to compromise the kernel through an in-kernel verification process. By running programs at the lowest level of the networking stack, via the eXpress Data Path (XDP [24]) hook, eBPF provides an excellent platform for developing and deploying NFs. The in-kernel verifier ensures memory, type, and resource safety; while being part of the kernel allows eBPF programs to reuse existing packet processing logic for the packets outside the scope of an NF. Also, being an inherent part of the kernel, eBPF makes deployment so much simpler. Large companies acknowledged the benefits coming from eBPF early on and are contributing to a large ecosystem of tools such as Cilium [5], Meta's Katran [9], and marketplaces (e.g., l3af [17]), thus further improving the development experience. Yet, these benefits come with performance limitations due to the kernel-based execution. Running inside the kernel implies interrupt-based packet processing and eliminates certain CPU instructions (e.g., SIMD) that offer significant performance benefits for packet processing. The in-kernel verifier prevents aggressive optimizations that could make verification harder or intractable.

So, there is a trade-off in deploying software NFs, which require simplicity, performance, and reliability [3] at scale: userspace solutions deliver raw performance at the expense of robustness, easy development, maintenance, and manageability, while in-kernel eBPF provides safety and ecosystem integration, but limits certain CPU features and compiler optimizations, harming performance.

This paper introduces uXDP[1], , a new runtime that uniquely bridges these two worlds by transparently moving verified eBPF NFs into userspace without recompiling. uXDP re-imagines the execution of NFs by blending the ease of deployment, verifier-driven safety, and ecosystem advantages of in-kernel eBPF with the flexibility and performance optimizations historically confined to

*Yusheng Zheng and Panayiotis Gavriil were previously at Imperial College London and any work related to this paper was done solely in affiliation with Imperial College London.

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

ACM ISBN 979-8-4007-2084-0/25/09
https://doi.org/10.1145/3748355.3748360

[1] uXDP is open-sourced in https://github.com/userspace-xdp

userspace. Even though running NFs in userspace prevents crashing the kernel, eBPF verification ensures that these NFs remain safe and reliable. Past work in verifying NFs [21, 41] also highlights the importance of such safeguards. uXDP embraces eBPF as the foundational programming model, directly accepting unmodified eBPF NF binaries, originally designed for kernel execution, and moving them into userspace. uXDP also introduces several compilation optimizations specifically for userspace eBPF runtimes. In doing so, uXDP not only improves performance but also enables XDP programs to run in environments where kernel eBPF is unavailable, extending the reach of the eBPF ecosystem into previously inaccessible scenarios.

We implement uXDP as a drop-in replacement runtime and evaluate it on a range of NFs, from simple packet filters to complex load balancers like Katran. With the same eBPF application binary, uXDP achieves throughput gains of up to 3.3× compared to in-kernel execution and improves Katran's performance by up to 40%. Crucially, uXDP delivers these benefits while preserving the trusted eBPF development model and integration workflows, making it easier for operators to adopt and scale across different environments.

The contributions of this paper are as follows:

(1) We present uXDP, a runtime that supports real-world, unmodified eBPF-based NFs in userspace, combining the ease of deployment, verification, and vast ecosystem of eBPF with the flexibility and performance of kernel-bypass frameworks.
(2) We develop optimization techniques for userspace eBPF runtimes that reduce execution overhead and improve the performance of complex NFs.
(3) We show the effectiveness of uXDP by showing significant performance benefits for various NFs.
(4) We discuss potential directions for future work in the deployment and optimization of eBPF.

2 Background and Motivation

This section introduces the background on eBPF/XDP, their verification and control plane support, highlights potential optimization opportunities, and examines existing userspace deployment approaches to clarify the motivating trade-offs.

2.1 eBPF Network Functions

eBPF: eBPF [13] allows safe, efficient user-defined code to run in the Linux kernel. Initially for packet filtering, it now supports many use cases and platforms, including userspace VMs like ubpf [12], rbpf [35], and eBPF for Windows [27]. Developers write restricted C, compile it to eBPF bytecode, which is verified [22], loaded via `bpf()`, attached to a kernel hook, and triggered on hook events. In-kernel verification imposes strict constraints, including control-flow safety (no unbounded loops or recursion), memory safety (no arbitrary, out-of-bounds or uninitialized memory reads), resource safety (no deadlocks or resource leaks) and no hardware exceptions (no division by zero). The verifier restricts the use of kernel helpers and kfuncs [14] to those with correct argument types, and requires type-correct, bounded references to eBPF maps.

NFs can be core components of production infrastructure serving hundreds of machines (e.g., load-balancer or firewall deployed at scale). Verification is crucial in production systems where robustness, safety, and maintainability are as important as raw speed. For example, a recent site-wide incident [38] at Bilibili, a leading Asian streaming service, demonstrated how an infinite loop in an API gateway can trigger major outages.

XDP: The eXpress Data Path (XDP)[24] enables eBPF NFs to process packets early in the network stack for near line-rate performance. Widely used for low-latency tasks like DDoS protection[7], load balancing [18], and monitoring [1], XDP runs in driver (native) or kernel (SKB) mode based on driver support.

eBPF-based NFs rely on a control plane application — a userspace process managing program installation, map configuration, policy updates, and runtime adjustments. For instance, Katran's [9] control plane dynamically updates eBPF maps to reflect changes in backend server states and reads the statistics map for health checks. NetEdit's [3] control plane orchestrates safe, dynamic eBPF-based NFs across thousands of services and millions of servers, offering unified abstractions, decoupled policies, and extensive testing. Control planes typically use libraries like libbpf [19] or libxdp [37]. However, even a basic tutorial-level XDP application such as `httpdump` demands hundreds of calls to over 15 syscalls and shared-memory operations for efficient management and communication—requirements that few userspace runtimes can meet without major improvements. This incompatibility challenge prevents complex eBPF programs from being adequately loaded and verified outside the kernel.

AF_XDP: To address some deployment complexity while retaining the performance benefits of userspace packet processing, the Linux community introduced the AF_XDP [23]. AF_XDP aims to combine DPDK-level performance with the ease of deployment of kernel-based networking. It employs a small XDP program to filter and redirect packets to the userspace before they enter the kernel's networking stack. Applications can access these packets through a new socket family, `AF_XDP` sockets. It supports two modes: *Copy mode* (higher overhead, broad compatibility) and *Zero-copy mode* (better performance, but hardware-dependent). However, by executing in userspace, developers still need to manage complexity and correctness themselves, with no built-in verification, safeguards, or standard policy mechanisms. Thus, achieving the same ecosystem and operational convenience as kernel-based solutions is non-trivial.

2.2 Optimization Margins in XDP

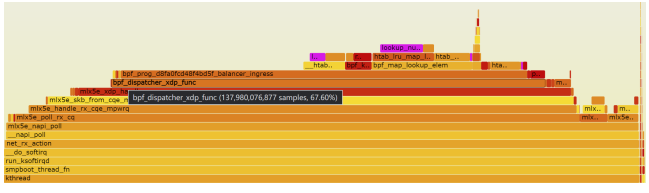

Figure 1: Katran flamegraph in driver mode

To identify optimization opportunities in XDP programs, we measure Katran's performance in Figure 1(methodology in subsection 5.1). In our study we observed the following. First, the eBPF program consumes roughly 67% of the CPU time, while the rest is

spent in the network driver, suggesting room for improvement in both the eBPF logic and the driver. Second, focusing on the eBPF program, about 50% of its CPU time is devoted to maps and helper calls. For Katran, 50% of packets result in an LRU cache hit, emphasizing the need for map and helper optimizations. On an LRU cache miss, Katran calculates a hash to determine packet direction, increasing CPU utilization. On a hit, it spends more time on map accesses and helper calls, as no hash computation is required.

3 Design

To enable userspace execution of eBPF-based NFs, we focus on existing XDP programs and design uXDP to meet the following three key requirements: **i)** remain compatible with existing XDP programs and their control planes to retain verification-based safety, ease of deployment, and benefit from the large eBPF ecosystem; **ii)** support different deployment models tailored to host capabilities; and **iii)** operate with or without access to the original NF source code.

3.1 Overview

uXDP consists of a control and a data plane process sharing memory for eBPF maps, bytecode, and metadata. It uses a userspace eBPF runtime with LLVM-based just-in-time (JIT) and ahead-of-time (AOT) compiling, while supporting a `DPDK` and an `AF_XDP` deployment modes. The control plane loads, verifies, and manages eBPF programs; the data plane executes them, enabling high-performance userspace processing while remaining compatible with existing eBPF programs. Figure 2 summarizes uXDP's two deployment modes and compares them to in-kernel XDP NFs (Figure 2A). Figure 2B illustrates the DPDK-based deployment, where two processes and the RX/TX rings run entirely in userspace. In the AF_XDP case (Figure 2C), RX and TX rings are shared with the kernel, requiring a kernel-resident XDP program to forward packets to the AF_XDP socket.

3.2 Optimizations

Running in userspace enables uXDP to easily implement optimizations that improve efficiency and reduce overhead. Although designed for uXDP, these optimizations could also benefit kernel eBPF with kernel changes (Section 7).

3.2.1 Inline Optimizations. Our analysis in subsection 2.2 shows that much of the eBPF program's execution time is spent on helper functions and map accesses. Therefore, uXDP inlines these calls during JIT/AOT compilation, applying optimizations directly to the eBPF bytecode without requiring the original C source.

Inline Helpers: Common helpers (e.g., data copying, `strcmp`, and `calc_csum`) are simple, do not interact with maps or other programs, and are implemented as function calls due to verifier constraints. By inlining them, uXDP avoids these calls and enables LLVM optimizations. uXDP provides LLVM IR implementations of these helpers, which it compiles and links with the lifted eBPF bytecode.

Inlining helper calls reduces instructions and unlocks further optimizations. It removes the function call overhead (register saves, stack manipulation) and benefits frequently used helpers like `bpf_xdp_load_bytes` and `bpf_xdp_adjust_head`. Inlining with constant arguments allows LLVM to perform constant folding and propagation, and dead code elimination to remove unnecessary checks. Inlined code inside loops enables loop unrolling, improving performance in cycle-intensive helpers like `bpf_csum_diff` — for example, when processing small IPv6 headers. Finally, exposing more instructions to the optimizer helps allocate registers more efficiently and improves overall performance.

Table 1: Inlined Helpers for XDP and eBPF

Helper Function	Description
`bpf_csum_diff`	Calculate checksum diff.
`bpf_xdp_load_bytes`	Load bytes from XDP frame.
`bpf_xdp_store_bytes`	Store bytes to XDP frame.
`bpf_strncmp`	Compare strings.
`bpf_xdp_adjust_tail`	Adjust XDP frame tail=.
`bpf_xdp_adjust_head`	Adjust XDP frame head.

Inline Maps: uXDP applies the same idea to map accesses. In kernel eBPF, maps are accessed via helpers, whereas uXDP inlines maps as global variables in the JITed LLVM IR. Array maps become directly accessible in memory as regular global variables without helper, while hash maps rely on inlined helpers. After deployment, global maps remain in shared memory for control plane access, and per-CPU maps have a copy for each CPU. uXDP allows programmers to mark inline maps as *frozen* (read-only `const` variables), enables constant folding and propagation. Instead of linking helpers at runtime, uXDP can generate a native ELF binary with maps and helpers included, easing deployment and allowing execution on devices without JIT and runtime dependencies.

Figure 3 shows uXDP's compile-and-run workflow, highlighting where inline optimizations occur (black arrows). The application, written in a high-level language, is first compiled to LLVM IR via `clang` (1) and then to eBPF bytecode (2), identical for both kernel and uXDP. At runtime, uXDP uses the kernel verifier to verify and lifts the bytecode to LLVM IR (3), applies optimizations (4), compiles it to native code (5), and executes the program (6).

3.2.2 LLVM IR Optimizations. When distributing and loading eBPF bytecode as described before, uXDP loses many optimization opportunities because eBPF bytecode discards valuable semantic information. Although eBPF bytecode is suitable for verification, it is not ideal for producing the most efficient native code. For instance, the eBPF backend uses a generic register assignment and a simplified data layout, and does not support architecture-specific instructions (e.g., *llvm.memcpy*). The type information may be lost when lifting eBPF bytecode to LLVM IR. When possible, uXDP uses the original LLVM IR from source instead of lifting it from eBPF bytecode. It enhances offline compilation by packaging this LLVM IR with the bytecode for distribution. At load time, uXDP uses the IR to generate optimized native code, while the bytecode is still used for verification.

The orange arrows in Figure 3 illustrate this alternative workflow. During the offline compilation, uXDP bypasses the compilation and lift steps by transforming and relocating directly on LLVM IR. uXDP transforms the LLVM IR to ensure it can access helpers by function call instead of address, and has the correct calling conventions

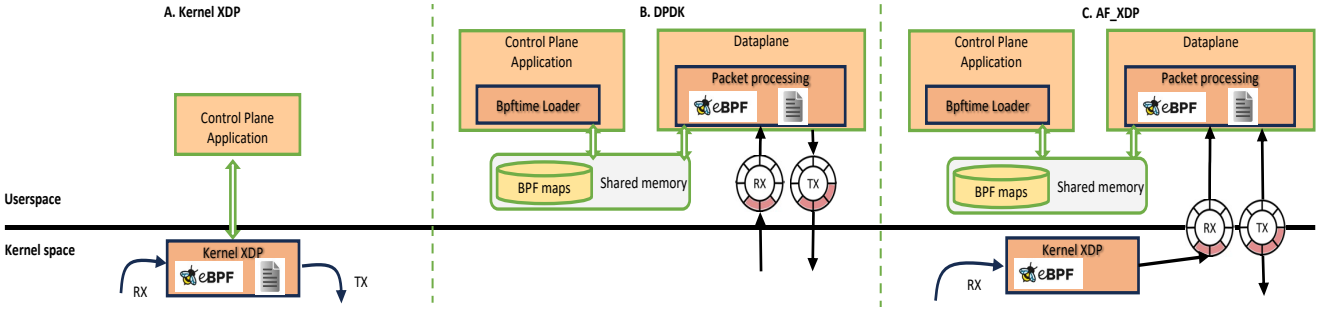

Figure 2: eBPF XDP program deployment in kernel and userspace enabled by uXDP using DPDK and AF_XDP. The file icon represents the XDP program.

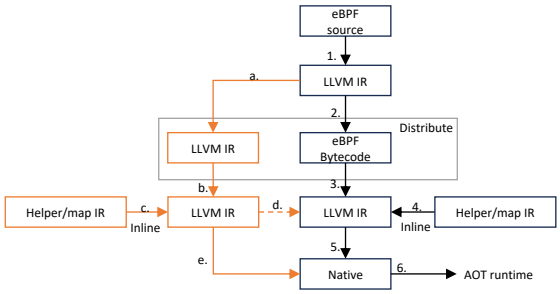

Figure 3: Compilation and deployment pipeline. The black pipeline does not assume access to LLVM IR. The orange pipeline describes the LLVM IR optimizations.

for execution in the eBPF runtime (a). The transformed LLVM IR is distributed along with the eBPF bytecode. At runtime, uXDP relocates the distributed LLVM IR to ensure it has access to maps with the correct map id (b), applies inline and other optimizations (c), and then JIT/AOT compiles and executes it (e).

To enhance safety, the LLVM IR can be signed by the toolchain. If the compiler is untrusted, tools like alive-tv [20] can verify it against IR lifted from eBPF bytecode. By bypassing eBPF bytecode during run time, uXDP achieves better register allocation, tailored data layouts, and richer type-based optimizations. Combined with userspace execution, this allows efficient native code leveraging SIMD and other hardware features not available in kernel eBPF.

4 Implementation

uXDP consists of two main parts: a userspace eBPF runtime and a compilation toolchain that implements the optimizations. uXDP is implemented with 3912 lines of C/C++ for the runtime and loader, 1274 lines of Python for optimization and loader scripts, and 524 lines of C for inline libraries.

eBPF Runtime: uXDP builds on `bpftime` [42], a userspace runtime designed for eBPF tracing programs. We chose `bpftime` because it: (i) supports loading and managing eBPF programs from a control plane application via the `bpftime` loader, (ii) uses shared memory for eBPF maps; and (iii) is currently the fastest eBPF runtime, leveraging LLVM for JIT/AOT compilation. However, `bpftime` does not support XDP when we start the project. uXDP extend the `bpftime` loader for the NF control plane and its runtime for the NF data plane. We extended `bpftime` to attach XDP programs to network interfaces using *bpf link* and added support for network-related maps (e.g. LPM_TRIE, LRU_HASH, ARRAY_OF_MAPS, DEVMAP) and helpers (bpf_xdp_*, bpf-_csum_diff). These patches have already been upstreamed.

A challenge was handling data structures like `xdp_md`, where `data` and `data_end` are 32-bit in kernel but require 64-bit pointers in userspace. While the kernel JIT handles this automatically, it's error-prone in userspace. uXDP solves this using CO-RE [28] and a custom BTF [15], allowing the same eBPF bytecode to run in both environments without recompilation.

Runtime Loader: When the control plane application loads an eBPF program, the runtime loader first verifies its safety, then either lifts the eBPF bytecode to LLVM IR or extracts LLVM IR directly from the eBPF ELF file, depending on the compilation path. It then runs LLVM optimization passes, producing a final LLVM IR representation of the eBPF program. Next, the runtime compiles this IR into native code using `opt-14` [34] and `llc-14` [33], optimizing for the target architecture. Finally, the loader supplies the native code to the userspace XDP runtime for packet processing.

Inline Optimization: uXDP provides a Python tool, `uxopt`, to perform inline optimizations (steps c or 4 in Figure 3). `uxopt` includes a pre-compiled LLVM IR library containing helper functions and implementations for `ARRAY` and `HASH_MAP`. It merges this library with the eBPF program LLVM IR and marks these helper functions as always inline, allowing the LLVM optimizer to apply aggressive inlining and other code improvements.

LLVM IR Optimization: uXDP modifies the offline compilation phase to enable LLVM IR optimizations. It introduces a tool called *uxcc*, which can replace `clang` in the build toolchain. `uxcc` uses `clang-14` [32] to produce unoptimized LLVM IR and then compiles optimized eBPF bytecode from it. During this process, it adjusts the IR for correct stack layout and helper name mangling, ensuring compatibility with the `bpftime` runtime. `uxcc` packages both the LLVM IR and the eBPF bytecode into a single ELF file and signs it for secure distribution. At deployment time, the uXDP loader verifies the eBPF bytecode for safety and checks the IR signature. It relocates maps within the LLVM IR, so helper calls use the correct map IDs. If the compiler is untrusted, uXDP can use Alive2 [20] to confirm that the included LLVM IR is functionally equivalent to the verified eBPF bytecode.

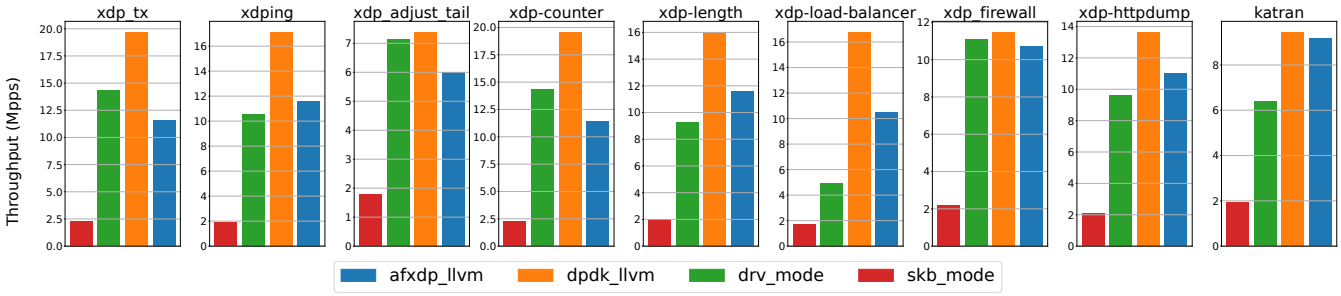

Figure 4: Throughput in Pkt/s for different kernel and userspace configurations for all the evaluated NFs.

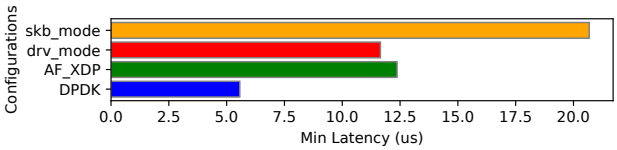

Figure 5: xdp_tx NF latency

5 Evaluation

In this section, we evaluate uXDP 's performance under different deployment configurations, measure optimization techniques impact, and assess the program portability provided by uXDP. Our evaluation seeks to answer the following questions:

(1) Is there any development cost in porting existing XDP programs to uXDP?
(2) What is the performance benefit of running XDP in userspace on DPDK or AF_XDP?
(3) What is the impact of the proposed optimization techniques on the performance of eBPF XDP-based NFs?

Table 2 summarizes the evaluated XDP programs, which come from the Linux kernel, Katran [9], and other open-source repositories [2, 11]. They cover a range of processing needs and complexities (from simple NFs to more sophisticated applications like Meta's Katran, and from firewalls and load-balancers to observability tools) allowing us to broadly demonstrate uXDP 's effectiveness. All programs run seamlessly in uXDP under all configurations without recompilation and pass the kernel verifier.

5.1 Methodology

Our testbed includes two servers with dual-port Mellanox ConnectX-6 Dx 100 Gbps NICs, connected back-to-back. Both use Intel Xeon Gold 5318N CPUs (24C/48T) on a single NUMA node (CPU 0–47), with 1.1 MiB L1d, 768 KiB L1i, 30 MiB L2, and 36 MiB L3 caches. One server (DUT) runs the NF; the other runs Pktgen [30] as the load generator. The DUT uses Linux kernel 6.7.10; the generator runs 6.3.4. Pktgen sends 64B TCP or 128B ICMP packets, which the DUT processes and returns for measurement. All eBPF programs run on a single core. As a baseline, we use XDP driver and skb modes, redirecting packets to one RX queue via `ethtool`. uXDP is tested with both DPDK and AF_XDP. AF_XDP uses RX queue 0, 4KB frame size, 64-packet batches, and enables `SO_PREFER_BUSY_POLL` and `XDP_USE_NEED_WAKEUP`. Linux uses `SCHED_OTHER` with priority 0. DPDK is configured with a 512 MB `mbuf` pool.

Name	Description	Source	Component	Insns
xdp_tx	Basic Mac-swap and TX	hXDP [4]	-	19
xdping	A ping (ICMP) server	Linux kernel sample	Helpers	79
xdp adjust tail	Generates ICMPv4 "packet too big" reply	Linux kernel sample	Helpers	151
xdp-counter	Control and count packets in array maps	Katran [9]	Array map	41
xdp-length	Summary packet lengths in hash map	Hand-crafted	Hash map	41
simple load balancer	A load balancer using array map and hash	Hand-crafted	Array map and helpers	167
blacklist firewall	A hash map L2 firewall and VRRP filtering	GitHub [2]	Per-CPU hash map	128
xdp-httpdump	Observe HTTP packet	Github [11]	Helpers, ring buffer	101
Katran	Meta's load balancer	Katran [9]	Array, hash and LRU map, helpers	2247

Table 2: Descriptions of various XDP programs evaluated. All pass the verification.

5.2 Deployment Versatility and its Impact

We focus on the throughput and latency impact of the different uXDP userspace deployment modes across various NFs. For this experiment, we use the fully optimized eBPF programs and assume access to source LLVM IR.

Figure 4 summarizes the throughput results in Pkts/s. We observe that DPDK maintains the highest performance across different network functions (NFs), given its poll mode driver. For simple eBPF NFs like `xdp-counter` and `xdp_adj-ust_tail`, the kernel drv_mode is faster than AF_XDP. For complex NFs like `xdp-load-balancer`, `xdping`, and `katran`, AF_XDP surpasses drv_mode due to the better-optimized code enabled by the userspace execution and produced by uXDP. This is a strong finding indicating that existing eBPF programs can achieve much more throughput while still depending entirely on technologies that are part of any

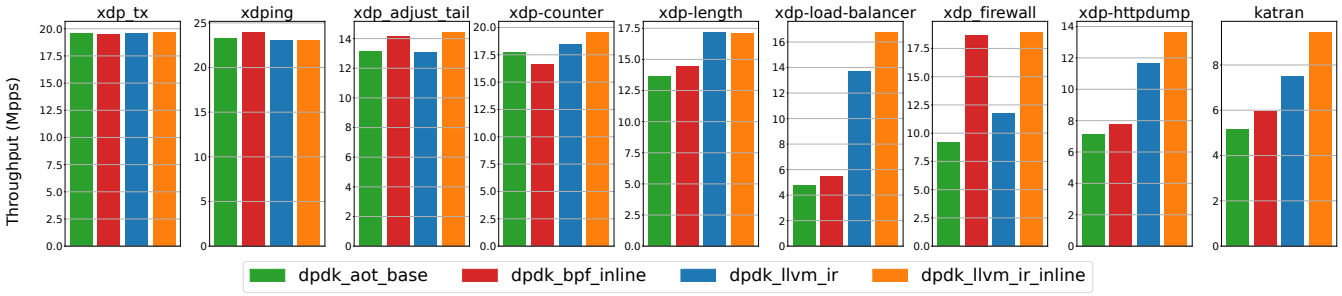

Figure 6: Impact of optimization techniques on eBPF XDP-based network functions. As in 3, (5) is for dpdk_aot_base, (4) is for dpdk_bpf_inline, (e) is for dpdk_llvm_ir, (c) is for dpdk_llvm_ir_inline.

Linux distribution, hence avoiding the deployment complexities of DPDK. skb_mode shows the lowest performance due to its higher processing overhead and the extra soft IRQ.

To evaluate latency, we deploy the simplest NF (`xdp_tx`) that swaps MACs and echoes packets, measuring unloaded round-trip latency. Figure 5 shows results: DPDK has the lowest latency, while AF_XDP has slightly higher latency than driver mode but better throughput in complex NFs due to userspace execution.

5.3 Optimization Techniques

To evaluate the optimizations impact, we use the best deployment setup from the previous section (DPDK) and compare the achieved throughput before and after applying them. Figure 6 shows packet rates with and without inlining, using the non-inlined version as the baseline across both compilation pipelines. In `dpdk_llvm_aot`, the bpftime AOT compiler converts eBPF instructions to LLVM IR, then uses opt [34] and llc [33] with O3 optimizations to compile IR to native code (i.e., no inline optimizations). `dpdk_bpf_inline` applies the inline optimizations on the LLVM IR lifted from eBPF byte code (step 4 in Figure 3). `dpdk_llvm_ir` represents the workflow that has access to the original LLVM IR (orange pipeline without inlining in Figure 3), while `dpdk_llvm_ir_inline` adds inlining.

The results show significant performance gains from additional optimizations, especially for complex NFs—e.g., Katran's throughput improved by 83%. The gap between the AOT baseline and kernel driver mode is mainly due to userspace helpers (via vDSO) and different map implementations in bpftime. In one case, inlining slightly reduced throughput (`xdp-counter` in `dpdk_bpf_inline`) due to increased branch prediction overhead.

6 Related work

There is prior work that focuses on out-of-kernel eBPF execution. Frameworks such as ubpf [12], rbpf [35], and DPDK's BPF library [31] execute eBPF programs in userspace, but they focus mainly on bytecode and lack support for essential NF components, such as maps, especially important for XDP-based NFs. Unlike these approaches, uXDP allows a safe and seamless transition to userspace execution. Parola et al. [29] compare userspace and in-kernel processing, highlighting AF_XDP's benefits in edge data centers. De Coninck et al. [6] extend eBPF to plugable protocols like PQUIC and xBGP, enabling network programmability outside of Linux. hXDP [4] implements an eBPF soft-core on FPGAs, and eHDL [36] generates hardware designs from eBPF/XDP.

Recent work also targets eBPF runtime optimizations. Mao et al. [25] present Merlin, a compile-time multi-tier optimization framework, while uXDP focuses on runtime JIT/AOT optimizations. Domain-specific optimizations include Miano et al. [26]'s in-kernel traffic control NFs and Xu et al. [39]'s K2, a synthesis-based compiler with formal guarantees but limited robustness. eNetSTL [40] implements certain common functionalities in native code. In contrast, uXDP provides an integrated compiler and runtime co-design, delivering substantial performance gains for eBPF XDP-based NFs independent of their business logic.

7 Future Work

In the current uXDP version, NFs can only drop or transmit packets (XDP_DROP, XDP_TX) but not forward them to the kernel. We plan to support re-injection into the kernel to enable broader use cases and compatibility with other network eBPF programs. Many of uXDP's optimizations, like inlining and SIMD usage, could also apply to kernel execution without breaking verifier guarantees. This would require kernel changes (e.g., saving FPU state), but based on our results, the performance gains may outweigh the additional overhead for complex NFs.

8 Conclusion

This paper introduces uXDP, a novel system that executes eBPF XDP-based network functions in userspace using kernel-bypass techniques and runtime optimizations. uXDP challenges the status of having to choose between "safe but limited" kernel solutions and "fast but fragile" userspace frameworks. By enabling a seamless transition from kernel to userspace without modifying the original eBPF code, uXDP offers enhanced safety, performance, and easy deployment, achieving up to 3.3× higher throughput for simple NFs and 40% better performance for Katran over kernel execution.

Acknowledgements

We thank the anonymous reviewers for their detailed and valuable feedback. This work is supported by a gift from Mysten Labs.

Ethics Statement

This work does not raise any entical considerations.

References

[1] Marcelo Abranches, Oliver Michel, Eric Keller, and Stefan Schmid. 2021. Efficient Network Monitoring Applications in the Kernel with eBPF and XDP. In *2021 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. 28–34. https://doi.org/10.1109/NFV-SDN53031.2021.9665095

[2] acassen. 2018. XDP FW: eXpress Data Path FireWall module. (2018). https://github.com/acassen/xdp-fw.

[3] 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 (ACM SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 721–734. https://doi.org/10.1145/3651890.3672227

[4] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. 2022. hXDP: Efficient software packet processing on FPGA NICs. *Commun. ACM* 65, 8 (2022), 92–100.

[5] cilium. 2025. eBPF-based Networking, Security, and Observability. (2025). https://github.com/cilium/cilium.

[6] Quentin De Coninck, Louis Navarre, and Nicolas Rybowski. 2024. On Integrating eBPF into Pluginized Protocols. *SIGCOMM Comput. Commun. Rev.* 53, 3 (feb 2024), 2–8. https://doi.org/10.1145/3649171.3649173

[7] Marinos Dimolianis, Adam Pavlidis, and Vasilis Maglaris. 2021. Signature-Based Traffic Classification and Mitigation for DDoS Attacks Using Programmable Network Data Planes. *IEEE Access* 9 (2021), 113061–113076. https://doi.org/10.1109/ACCESS.2021.3104115

[8] DPDK. 2025. Data Plane Development Kit. (2025). https://github.com/DPDK/dpdk.

[9] facebookincubator. 2018. A high performance layer 4 load balancer. (2018). https://github.com/facebookincubator/katran.

[10] fd.io. 2018. Vector Packet Processing. (2018). https://docs.fd.io/vpp/18.11/.

[11] hamidrezakhosroabadi. 2025. A simple xdp application to observe tcp connections in userspace. (2025). https://github.com/hamidrezakhosroabadi/xdp-observer.

[12] iovisor. 2025. Userspace eBPF VM. (2025). https://github.com/iovisor/ubpf.

[13] Linux kernel maintainers. 2025. BPF document. (2025). https://docs.kernel.org/bpf/index.html.

[14] Linux kernel maintainers. 2025. BPF Kernel Functions (kfuncs)¶. (2025). https://docs.kernel.org/bpf/kfuncs.html.

[15] Linux kernel maintainers. 2025. BPF Type Format (BTF). (2025). https://docs.kernel.org/bpf/btf.html.

[16] Linux kernel maintainers. 2025. PCI Express I/O Virtualization Howto. (2025). https://docs.kernel.org/PCI/pci-iov-howto.html.

[17] l3af maintainers. 2025. Complete lifecycle management of eBPF programs in the kernel. (2025). https://l3af.io/.

[18] Jung-Bok Lee, Tae-Hee Yoo, Eo-Hyung Lee, Byeong-Ha Hwang, Sung-Won Ahn, and Choong-Hee Cho. 2021. High-Performance Software Load Balancer for Cloud-Native Architecture. *IEEE Access* 9 (2021), 123704–123716. https://doi.org/10.1109/ACCESS.2021.3108801

[19] libbpf Authors. 2025. Libbpf. (2025). https://libbpf.readthedocs.io/en/latest/libbpf_overview.html.

[20] Nuno P Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: bounded translation validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 65–79.

[21] Dana Lu, Boxuan Tang, Michael Paper, and Marios Kogias. 2024. Towards Functional Verification of eBPF Programs. In *Proceedings of the ACM SIGCOMM 2024 Workshop on eBPF and Kernel Extensions*. 37–43.

[22] BPF maintainers. 2025. eBPF verifier. (2025). https://docs.kernel.org/bpf/verifier.html.

[23] Linux maintainers. 2025. AF_XDP. (2025). https://www.kernel.org/doc/html/next/networking/af_xdp.html.

[24] Linux maintainers. 2025. XDP - eXpress Data Path. (2025). https://prototype-kernel.readthedocs.io/en/latest/networking/XDP/.

[25] Jinsong Mao, Hailun Ding, Juan Zhai, and Shiqing Ma. 2024. Merlin: Multi-tier Optimization of eBPF Code for Performance and Compactness. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 639–653.

[26] Sebastiano Miano, Xiaoqi Chen, Ran Ben Basat, and Gianni Antichi. 2023. Fast In-kernel Traffic Sketching in eBPF. *SIGCOMM Comput. Commun. Rev.* 53, 1 (apr 2023), 3–13. https://doi.org/10.1145/3594255.3594256

[27] microsoft. 2025. eBPF for Windows. (2025). https://github.com/microsoft/ebpf-for-windows.

[28] nakryiko. 2025. BPF CO-RE reference guide. (2025). https://nakryiko.com/posts/bpf-core-reference-guide/.

[29] Federico Parola, Roberto Procopio, Roberto Querio, and Fulvio Risso. 2023. Comparing User Space and In-Kernel Packet Processing for Edge Data Centers. *SIGCOMM Comput. Commun. Rev.* 53, 1 (apr 2023), 14–29. https://doi.org/10.1145/3594255.3594257

[30] pktgen. 2025. DPDK based packet generator. (2025). https://github.com/pktgen/Pktgen-DPDK.

[31] DPDK Project. 2023. BPF Library. (2023). https://doc.dpdk.org/guides/prog_guide/bpf_lib.html Accessed 2025-07-22.

[32] LLVM project. 2025. Clang 14.0.0 documentation. (2025). https://releases.llvm.org/14.0.0/tools/clang/docs/ReleaseNotes.html.

[33] LLVM project. 2025. llc - LLVM static compiler. (2025). https://llvm.org/docs/CommandGuide/llc.html.

[34] LLVM project. 2025. opt - LLVM optimizer. (2025). https://llvm.org/docs/CommandGuide/opt.html.

[35] qmonnet. 2025. Rust virtual machine and JIT compiler for eBPF programs. (2025). https://github.com/qmonnet/rbpf.

[36] Alessandro Rivitti, Roberto Bifulco, Angelo Tulumello, Marco Bonola, and Salvatore Pontarelli. 2023. eHDL: Turning eBPF/XDP Programs into Hardware Designs for the NIC. 208–223. https://doi.org/10.1145/3582016.3582035

[37] xdp project. 2025. xdp-tools - Library and utilities for use with XDP. (2025). https://github.com/xdp-project/xdp-tools.

[38] XRay. 2025. Resolving Bilibili's major site incident with OpenResty XRay. (2025). https://medium.com/@openresty/resolving-bilibilis-major-site-incident-with-openresty-xray-70ae6e1d875a.

[39] Qiongwen Xu, Michael D. Wong, Tanvi Wagle, Srinivas Narayana, and Anirudh Sivaraman. 2021. Synthesizing safe and efficient kernel extensions for packet processing. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 50–64. https://doi.org/10.1145/3452296.3472929

[40] 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 *Proceedings of the Twentieth European Conference on Computer Systems (EuroSys '25)*. Association for Computing Machinery, New York, NY, USA, 42–58. https://doi.org/10.1145/3689031.3696094

[41] Arseniy Zaostrovnykh, Solal Pirelli, Rishabh Iyer, Matteo Rizzo, Luis Pedrosa, Katerina Argyraki, and George Candea. 2019. Verifying software network functions with no verification expertise. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 275–290. https://doi.org/10.1145/3341301.3359647

[42] Yusheng Zheng, Tong Yu, Yiwei Yang, Yanpeng Hu, XiaoZheng Lai, and Andrew Quinn. 2023. bpftime: userspace eBPF Runtime for Uprobe, Syscall and Kernel-User Interactions. (2023). arXiv:2311.07923