eNetSTL: Towards an In-kernel Library for High-Performance eBPF-based Network Functions

Bin Yang[1], Dian Shen[1*], Junxue Zhang[2*], Hanlin Yang[1], Lunqi Zhao[1], Beilun Wang[1], Guyue Liu[3], Kai Chen[2]
[1]Southeast University
[2]Hong Kong University of Science and Technology [3]Peking Univeristy

Abstract

Using extended Berkeley Packet Filter (eBPF) to implement networking functions (NFs) has been a promising trend for modern network infrastructure. In this paper, we endeavor to implement 35 representative NFs with eBPF, but encounter inherent problems of either incomplete functionality or performance degradation of up to 49.2%. Conventional solutions like modifying the eBPF infrastructure or implementing functions directly in the kernel can lead to intrusive and unstable modifications.

To address these challenges, we present eNetSTL, the first in-kernel library for eBPF-based network functions. At its core, eNetSTL identifies shared performance-critical behaviors among the 35 NFs, and abstracts these behaviors into a minimal and stable set of in-kernel components (containing a memory wrapper, three algorithms, and two data structures). It reduces interaction overhead with eBPF and mitigate safety risks by using Rust and a metadata-assisted verifier. By doing so, eNetSTL minimizes intrusions into the kernel space, ensuring stability and compatibility with current and future requirements of eBPF-based NFs. We demonstrate the capabilities of eNetSTL by presenting three real-world use cases that leverage its comprehensive functionalities. Extensive testbed experiments on seven categories of NFs show that their implementation with eNetSTL outperforms the eBPF counterparts by up to 1.8×, in terms of packet processing rate.

CCS Concepts: • **Networks → Middle boxes / network appliances**; • **Software and its engineering → Software libraries and repositories**.

*Corresponding authors.

Keywords: Network functions, eBPF, Performance profiling, In-kernel library

ACM Reference Format:
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 *Twentieth European Conference on Computer Systems (EuroSys '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3689031.3696094

1 Introduction

Software network functions (NFs) are crucial components of modern network infrastructure. For example, in the cloud environment, data plane communications between virtual nodes require virtual switches, virtual routers, and other network functions, where their efficient operations are essential for ensuring the high performance and reliability of network services. A recent trend is to implement these network functions using the extended Berkeley Packet Filter (eBPF), as evidenced by existing research [1, 29, 51–54, 62, 79], open-source projects [14, 39, 57], and the deployment of eBPF-based NFs in production by well-known companies such as Meta [57], Google Cloud [31], and CloudFlare [49]. This trend can be attributed to eBPF's flexibility, maintainability, high performance, and good integration into cloud infrastructure [7, 60, 73].

In this paper, we intensively survey 35 representative works [3, 5, 6, 8, 10, 15, 20, 22–27, 34–36, 44–47, 50, 55, 58, 59, 61, 63, 64, 66–68, 72, 74, 80–82] covering various categories of popular NFs, including key-value query, membership test, packet classification, load balancing, counting, sketching, and queuing. We attempt to implement their core operations[1] using eBPF. However, we encounter two problems: (1) Incomplete functionalities: it is unable to implement three of them due to the inherent programming limitations in the non-contiguous memory of eBPF. (2) Performance degradation: while 28 operations can be implemented using eBPF, they suffer from significant performance degradation, reaching up to 49.2%, compared to their in-kernel counterparts. For instance, the absence of SIMD instructions results in a

[1]For example, in Cuckoo Switch [82], we implement the FIB lookup using eBPF. For interested readers, we provide details of our eBPF implementations in the supplementary material.

21.5-29.8% performance reduction in key-value query and a 19.2-49.2% decrease in sketching. Additionally, the overhead from eBPF safety mechanisms leads to a performance decline of 14.8%-31.6% in queuing.

There are two solutions for the aforementioned problems. However, they either cause substantial intrusion into the kernel code base or suffer from frequent updates of the kernel modules, compromising the gain of eBPF technology. Specifically, the first solution is to redesign the eBPF infrastructure [28, 38], including extensions to the eBPF ISA and modifications to the verifier [38]. However, it causes a dramatic intrusion into the kernel and presents challenges for prompt and practical deployment to address immediate community needs [52]. For example, expanding the eBPF instruction set requires modifying the JIT (just-in-time) compiler of as many as 14 architectures [4], and modifying the eBPF verifier raises concerns regarding the correctness of the verifier itself [65]. The second solution is to implement each NF as a kernel module and integrate them into the kernel as needed. However, with the existence of large amounts of NFs and their variants and their rapid advancements in the network community, the kernel experiences instability due to the frequent replacement of kernel modules, posing challenges in terms of maintainability.

In this paper, we ask: *can we achieve comprehensive functionalities and high performance for eBPF-based network functions without compromising eBPF's advantages of flexibility and low intrusion?* To answer this question, we dive into the design of 35 representative works and observe that they share six similar behaviors, which considerably determine the overall performance (§3), thus presenting opportunities for answering the question.

Based on the above observation, we take the initiative in this direction by proposing eNetSTL, the first standard in-kernel library for eBPF-based networking functions[2]. It is designed to be *spatially minimal*—minimizing intrusion into the kernel—and *temporally stable*—maintaining kernel stability to accommodate current and future network functions. The core design of eNetSTL lies in providing appropriate abstractions of performance-critical operations of NFs , considering the interaction overhead with eBPF while not compromising its safety guarantee. Specifically, to support non-contiguous memory in eBPF without compromising the safety guarantee, we develop a memory wrapper with proxy-based memory management and lazy safety checking (§4.2). Second, to support high-performance operations, we trade excessive generality for improved performance by designing higher-level interfaces that reduce interaction overhead while still satisfying the requirements of involved network functions (§4.3). Finally, to mitigate the safety risks associated with using the in-kernel library, we implement eNetSTL in Rust to enhance memory safety and conduct a targeted review of the remaining unsafe code for other critical safety properties, including the correctness of safe abstractions and safe termination. Additionally, we leverage metadata to assist the verifier in ensuring safe interactions between eBPF and eNetSTL. While all 35 works have already benefited from eNetSTL, we believe that eNetSTL could also be applied to new research works in the future since they may share many similarities with these 35 representative works.

We then demonstrate the comprehensive functionality of eNetSTL via case studies. Due to space limitations, we select three representative case studies to illustrate the detailed usage of eNetSTL for eBPF-based network functions. Furthermore, to compensate the small set of detailed case studies, we perform testbed experiments with 11 network functions, covering all types in Table 1. In detail, to demonstrate that eNetSTL enables network functions requiring non-contiguous memory, we implement the key-value query operation based on skip-list in NFD-HCS [47] using the memory wrapper. Evaluation results show that eNetSTL not only fully enables this function but achieves comparable performance as the kernel implementation with a gap less than 8.54%. Moreover, the evaluations of the remaining network functions demonstrate that eNetSTL can support high-performance NFs through its designed algorithms and data structures. Experimental results show that eNetSTL achieves 14.6% to 75.4% higher packet processing rates over its eBPF counterparts. And the performance of eNetSTL is close to the in-kernel implementation with a negligible gap of 3.42% on average and 5.24% at most.

As a final note, while eBPF technology may advance in the future, in this paper, we target to provide a plug-and-play library that can immediately benefit the community. Moreover, we believe, for future works, if not directly applicable with eNetSTL, our idea of extracting shared performance-critical behaviors should also be beneficial. All source codes of eNetSTL and the use cases are open-sourced in GitHub[3].

2 Background and Motivation

2.1 eBPF-based Network Functions

Software network functions plays a significant role in modern networking infrastructure. We survey seven categories of NFs that are widely utilized in practice, including key-value query [27, 44, 47, 59, 82], membership test [8, 10, 25, 26, 34, 36, 61], packet classification [67, 68, 74], load balancing [20, 23, 58], counting [3, 5, 6, 22, 50, 55, 81], sketching [15, 35, 45, 46, 80], and queuing [24, 63, 64, 66, 72]. To satisfy various requirements, how to efficiently and flexibly implement these functions on demand is a key problem for today's networking infrastructure.

As an emerging kernel extension technology, the extended Berkeley Packet Filter (eBPF) combined with in-kernel hooks

[2]eNetSTL for eBPF-based network functions can be analogously compared to the Standard Template Library (STL) for C++

[3]https://github.com/chonepieceyb/eNetSTL

such as XDP [33, 76] and TC is a promising solution. Network functions implemented with eBPF are widely deployed in production by well-known companies [7, 31, 49, 57], becoming fundamental building blocks in today's cloud infrastructure [7]. For example, Meta put eBPF into production at scale with its load balancer project Katran [57] and Google Cloud currently uses an eBPF-based network dataplane [31]. Beyond that, there are many research [1, 29, 51–54, 62, 79] and open-source projects [14, 39, 57], which implement network functions based on eBPF. The trend towards eBPF is due to its significant advantages. eBPF integrates efficiently with current cloud ecosystems. For instance, in intra-host container communication, the in-kernel XDP data path outperforms DPDK datapath in OvS [73]. Additionally, eBPF supports high-performance packet processing without saturating the CPU, enabling NF and non-NF applications to run on the same device without performance penalties brought by kernel-bypass solution (e.g. DPDK) [57, 60, 62]. Moreover, eBPF allows dynamic loading of user code for safe execution without kernel source modifications or frequent updates, enhancing maintainability and flexibility, and enabling faster development and deployment of demanding network functions [53].

To implement NFs using eBPF, users develop customized programs using the high-level C language and utilize the Clang/LLVM toolchain to compile them into eBPF bytecode. Subsequently, users load these bytecode into the kernel through system calls. During the loading process, the eBPF verifier conducts static analysis on the bytecode, ensuring their safety when running NFs in the kernel. The current eBPF verifier adopts a very strict rule and will reject any code that violate the rules. We suggest interested readers refer to the paper [77] for more information about the verifier. Upon successful verification, the eBPF program is Just-In-Time (JIT) compiled into machine-specific instructions and is attached dynamically to a specific hook, such as the XDP hook, for efficient execution.

2.2 Problems Identified

In this paper, we take the initiative to implement the core operations of the 35 real-world NFs using eBPF. However, we encounter two significant problems in our trial. First, three of them cannot be implemented with eBPF due to inherent programming limitations (P1). Second, 28 of them exhibit performance degradation ranging from 14.8% to 49.2% compared to in-kernel implementation due to various factors, such as the absence of specific instructions in the eBPF ISA (P2). Only four of them can be implemented using eBPF without performance degradation. Worth noting, our pure-eBPF implementations leverage existing eBPF foundational mechanisms (e.g., BPF map [70] and BPF helper [71]) and advanced ones (e.g., BPF kfunc [69] and BPF kptr [17]). Table 1 summarizes the 35 works along with our identified problems in their eBPF implementation. We then provide a detailed explanation as follows.

P1: Incomplete functionalities. eBPF imposes strict limitations on the use of non-contiguous memory, preventing the implementation of core components of certain NFs, such as key-value query based on skip-list [47] and queuing [24] based on red-black tree[4]. Using non-contiguous memory needs support for a variable number of dynamic memory to be persisted for later use. Despite recent Linux kernels [40] (version 6.1 or later) supporting allocating dynamic memory and persisting it into a BPF map, the verifier enforces a predefined and fixed number of dynamic memories that can be persisted. Thus, non-contiguous memory is unattainable in the existing eBPF due to the limitation of no support for variable dynamic memories.

P2: Performance degradation. Firstly, the eBPF RISC instruction set lacks support for specific instructions, including SIMD (Single Instruction, Multiple Data) instructions and bit manipulation instructions, resulting in performance degradation. For example, in sketching, the absence of SIMD instructions may result in a 49.2% performance degradation due to inefficient computation of multiple hash functions. Additionally, the absence of bit manipulation instructions in Eiffel [64], which relies on the FFS (find the first set) instruction for rapid queuing, results in performance degradation of 14.8%. Secondly, the safety mechanisms of eBPF require coupling between BPF link lists and BPF spin-locks, notably impacting NFs utilizing link lists to store entries, such as the performance degradation of 27.1% in Carousel [63]. Lastly, the invocation of eBPF helpers, such as `bpf_get_prandom_u32`, leads to a 46.6% performance decrease on average when used on a per-packet basis [3, 6, 52].

2.3 Conventional Solutions & Their Problems

To address the above problems, two conventional solutions can be considered. The first solution is modifying the overall architecture of eBPF, for example, extending the instruction set of eBPF, enhancing the verifier, introducing new runtime and language-level safety mechanisms [38], and decoupling the verification process from kernel to user space [28]. However, this approach is impractical due to its extensive modifications to the kernel. For instance, expanding the eBPF instruction set requires modifications to the architecture-specific JIT (just-in-time) compiler within the kernel codebase, currently spanning up to 14 hardware architectures. Furthermore, extending the instruction set mandates modifications to the verifier's code, given that the verifier conducts validation at the level of eBPF instructions. Nevertheless, altering the verifier introduces new challenges in ensuring

[4]Currently, eBPF implements red-black tree [16] and link list [40] as kernel utilities. However, it still cannot fulfill the demands for network functions involving fully customized non-contiguous memory layouts, such as key-value query based on skip list.

NFs	Representative works	Performance degradation with eBPF
Key-value Query	Cuckoo Hash [59], Cuckoo Switch [82], d-ary Cuckoo [27], SILT [44], NFD-HCS (×) [47]	$\downarrow$ 21.55% – ∞,
Membership Test	Bloom Filter [8], Summary Cache [26], Cuckoo Filter[25], d-left BF [10], Rank-Indexed [34], Blocked BF [61], VBF [36]	$\downarrow$ 15.7% – 27.7%
Packet Classification	Hypercut (✓) [67], Efficut (✓) [74], TSS [68]	$\downarrow$ 0 – 33.8%
Load Balancing	Maglev (✓) [23], Beamer (✓) [58], EFD [20]	$\downarrow$ 0 – 37.9%
Counting	Space Saving (×) [50], CSS [5], HSS [55], RHSS [6], Tiny Table [22], Memento [3], HeavyKeeper [81]	$\downarrow$ 25.6% – ∞
Sketching	Count-min Sketch [15], UnivMon [46], Sketch Visor [35], Elastic Sketch [80], Nitro Sketch [45]	$\downarrow$ 19.2% – 49.2%
Queuing	Eiffel [64], PCQ [66], Carousel [63], Non-cascade TW [72], FQ/pacing (×) [24]	$\downarrow$ 14.8% – ∞

Table 1. Investigation of representative works of NFs. When implementing their core components with eBPF, three of them are unable to implement (indicated by ×), 28 of them suffer from performance degradation ranging from 14.8% to 49.2%, and only four of them can be properly implemented (indicated by ✓).

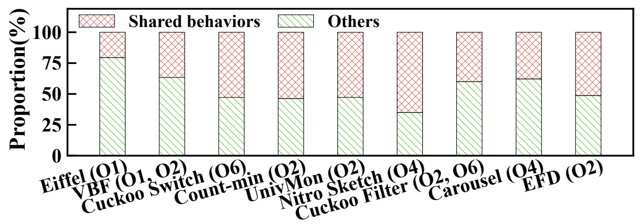

Figure 1. The performance proportion of observed common behaviors in NFs, in terms of executing time, ranges from 20.6% to 65.4%. The Ox represents the x-th observation.

the correctness of the modified verifier [65]. While theoretically feasible, the proposal to redesign the safety and programming architecture of eBPF is far from being practically deployed because it may have negative impacts on numerous existing eBPF-based network functions in the industry. Beyond the significant intrusion into the kernel, modifying the eBPF ISA could undermine eBPF's portability across multiple architectures, as not all of them support advanced instructions. For instance, when a recent patch proposed adding memory barrier instructions in the ISA, the community prefers to implement the memory fence using bpf kernel functions (kfuncs) [2].

The second solution involves implementing all functionality-unattainable and performance-degrading NFs as kernel modules and integrating them into the kernel. However, integrating all NFs into the kernel will impose a substantial intrusion into the kernel, while integrating individual NFs on demand may result in frequent kernel module replacements as demand changes, leading to two problems of maintainability and safety. Firstly, such per-NF per-LKM approach leads to an increased number of modules. Each kernel module requires additional code to ensure compatibility with different kernel versions, which reduces maintainability [73]. Secondly, ensuring kernel module safety requires additional code audits. The per-NF per-LKM approach necessitates reviewing each NF module individually, which increases safety risks since kernel modules do not have an inherent safety guarantee. Given the swift development in the networking community, such "one module for each" method could render the kernel quite unstable. In essence, this approach lacks flexibility and generality to support future NFs.

3 Observations & Opportunities

In order to solve the problems without causing dramatic intrusion and instability to the kernel, we further take a deeper look at the design of 35 representative works (Table 1) and observe that they share similar performance-critical behaviors. This presents an opportunity for a new practical solution that minimizes intrusion into the kernel to meet immediate requirements from the community and keeps the kernel stable for both current and future network functions. We highlight six shared behaviors as follows.

1. Leveraging hardware bit instructions. Membership test [34, 36], counting [22], sketching [46], and queuing [64, 72] operations share the behavior of encoding position information compactly in bitmaps (i.e., 64-bit unsigned long) and leverage hardware bit manipulation instructions (e.g., FFS and POPCNT) for fast searching and locating to avoid costly software-based queries. For example, in the queuing operations based on buckets of Eiffel [64], the information about which bucket contains elements can be encoded into a bitmap (i.e., bit `i` is set to one if `buckets[i]` contains elements), and FFS (find the first Set) instruction can be leveraged to achieve $O(\lceil \frac{n}{64} \rceil)$ (If a single ffs instruction operates on 64-bit data) lookup time by locating the index of the first set bit of a u64 in three CPU cycles [64].

2. Using multiple hash functions. Key-value query [27], membership test [8, 26], load balancing [20], counting [81], and sketching [15, 35, 45, 46, 80] operations use multiple hash functions to distribute elements across different locations and reduce collision. For example, given a key, Count-min sketch [15] determines the locations of d distinct counters to be updated or aggregated with d distinct hash. Moreover, key-value query operation based on d-ary Cuckoo hash [27] extends the original algorithm by utilizing d hash functions to determine the d possible positions where a key might be stored to reduce collision.

3. Building on fundamental data structures. Sketching [35, 80] and queuing [63, 64, 66, 72] operations are built upon fundamental data structures, notably, the counter-based heap and link list. For example, counter-based heaps are commonly utilized to construct a fast path for handling elephant flows in sketches [35, 80]. And constant-time queuing operations [64, 66, 72, 75] are designed based on the idea of bucket sorting and utilize linked lists to store the underlying elements.

4. Updating based on a random number. Probabilistic updating is shared in counting [3, 6] and sketching [45] operations to reduce the overhead of the update process. However, generating per-packet random numbers using BPF helper functions (i.e., *bpf_get_prandom_u32*) introduces significant overhead.

5. Operating on non-contiguous memory spaces. Key-value query [47], counting [50], and queueing [24] operations adopts non-contiguous memory. Given the dynamic and bursty nature of network traffic, it is challenging to predict in advance the length of required memory for handling dynamic flows. Over-allocating too much contiguous space leads to memory waste while less-allocating results in packet loss. In such scenarios, it is more appropriate to allocate and address the memory on demand, resulting in non-contiguous memory spaces.

6. Arranging multiple buckets in contiguous memory. In some other scenarios, high-performance operations work only on contiguous memory space, e.g., key-value query [44, 82], membership test [10, 25, 61], packet classification [68], and counting [5, 6, 22, 55, 81]. They tend to arrange multiple buckets in contiguous memory to facilitate fast counting or handle collision. Typically, it iterates and compares a key to the contents of the buckets (comparison) or locates the first minimum/maximum value within the buckets (min/max reduction). For example, in the key-value query operation utilized in Cuckoo Switch [82], a blocked Cuckoo hash [19] is used and its entries indexed by hashing are expanded into an array with a fixed number of buckets to mitigate collisions. Subsequently, the key or signature is compared with those stored in each bucket to retrieve the value.

To quantitatively understand how these shared behaviors impact the overall performance, we further conduct testbed experiments to measure the execution time of these operations. We evaluate the throughput of network functions with and without above behaviors under the same traffic. The results are shown in Figure 1 and we observe that these operations take a considerable portion of the overall execution time, ranging from 20.6% to 65.4%. As a result, optimizing these shared parts plays a key role in optimizing the end-to-end performance of NFs, thus, worth investigating. However, it's challenging to optimize them in eBPF due to inherent limitations of eBPF itself. Specifically, the absence of SIMD instructions makes optimizing behaviors of using multiple hash functions and arranging multiple buckets difficult. The lack of bit manipulation instructions hinders optimization for leveraging hardware bit instructions. The mandatory use of linked lists and spin locks by eBPF affects the behavior of using linked lists as a fundamental data structure, and

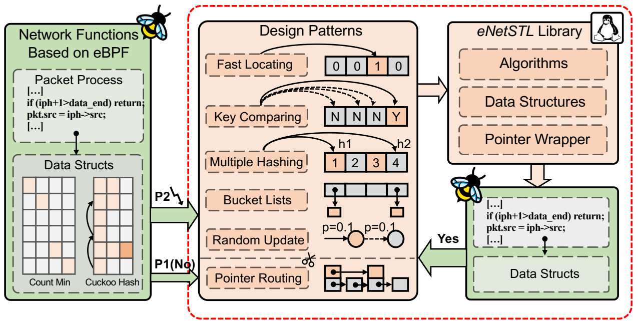

Figure 2. eNetSTL Design. The eBPF-based NFs share six identified performance-critical operations. eNetSTL abstracts them into a library.

the costly eBPF helper function to generate random numbers impacts the behavior of updating based on a random number. It should be noted that the behavior of operating on non-contiguous memory spaces is not shown in Figure 1, since eBPF does not support it and such issue restricts a considerable amount of NFs to benefit from eBPF. As a result, an alternative solution should be explored.

4 eNetSTL

Based on the above observations, we propose eNetSTL, the first in-kernel library for eBPF-based network functions. On one hand, eNetSTL aims to fully utilize the existing eBPF infrastructure, thus minimizing the intrusion to the kernel. On the other hand, the components in eNetSTL are carefully abstracted to achieve functional stability. When implemented as a loadable kernel module, eNetSTL can be readily deployable for existing 35 eBPF-based networking functions and does not suffer from frequent updates when accommodating future requirements. Specifically, eNetSTL contains one wrapper, three algorithms, and two data structures. The key design lies in abstracting suitable components to simultaneously fulfill three crucial yet conflicting goals of generality, performance and safety. In achieving the design goal, we encounter two challenges here:

- How to support non-contiguous memory in eBPF without compromising the safety guarantee provided by eBPF.
- How to design interfaces for specific algorithms and data structures to minimize the interaction overhead with eBPF while preserving generality.
- How to mitigate safety risks of eNetSTL itself, especially concerning safe termination and memory safety.

To address the two challenges: (1) We develop a memory wrapper with proxy-based memory management and lazy safety checking to effectively support non-contiguous memory in eBPF. (2) To achieve high performance, we design interfaces based on the generality required by network functions, avoiding the use of low-level interfaces that introduce interaction overhead but provide unnecessary generality.

(3) We employ Rust and code review to enhance the safety of individual APIs and facilitate safe interaction between eBPF and eNetSTL by providing metadata to the verifier. The design overview of eNetSTL is shown in Figure 2.

4.1 Background of Kfunc and Kptr

eNetSTL extensively leverages eBPF's kfunc[69], kptr [17], and associated verifier functionalities. This section provides an overview of the technical background.

BPF Kernel Functions (kfunc) [69] allow new kernel functionalities to be dynamically exposed to eBPF programs at runtime via kernel modules, without needing recompilation, kernel reinstalls, or system restarts. A crucial aspect of exposing a kfunc is specifying its metadata. The static verifier ensures safety by validating that eBPF programs adhere to the metadata defined by the developer, rather than verifying the implementation of the kernel functions themselves. There are three specific scenarios: firstly developers can specify which types of BPF programs are permitted to use the kfunc. Secondly, developers can annotate function arguments by suffixing variable names in the kfunc's prototype. For example, the `int val__k` annotation as k ensures that this argument is a constant int. Thirdly, developers can annotate the kfunc itself to guide the verifier. For example, annotating the kfunc with `KF_RET_NULL` indicates that the pointer returned by the kfunc may be NULL. Hence, it forces the user to do a NULL check on the pointer returned from the kfunc before using it.

The BPF kernel pointer (kptr) [17] is an abstraction for kernel resources to facilitate the safe interaction between the eBPF program and kernel. eBPF verifier relies on the three rules to prevent resource leaks and use-after-free issues in eBPF programs. Firstly, eBPF programs use kfuncs (annotated as `KF_ACQUIRE`) or special helper functions to allocate new kptrs. Alternatively, they can retrieve existing kptrs from BPF maps (using the `bpf_kptr_xchg` helper function). The verifier marks the newly allocated or obtained kptr as invalid until it is explicitly checked as not NULL. Secondly, only valid kptrs can be safely passed to the kfunc as arguments or be dereferenced to prevent use-after-free. Thirdly, to avoid resource leaks, the eBPF verifier ensures that a kptr that is allocated (or obtained from BPF maps) must be released by helper function or kfunc annotated as `KF_RELEASE` (or persisted to BPF maps using `bpf_kptr_xchg`).

4.2 Proxy-based and Lazy Memory Wrapper

The difficulties of using non-contiguous memory in eBPF-based NFs stem from the need to support both unpredictable length and persistent storage of the memory. To achieve this without compromising the eBPF safety guarantee, we design a memory wrapper employing proxy-based memory management and lazy safety checking. The memory wrapper consists of a set of functions and APIs for usage in eBPF.

Proxy-based memory ownership management. The current eBPF does not support the persisted and unpredictable length of memory due to the requirement that the number of dynamically allocated memories persisting in the BPF map should be known in advance and fixed [17][5].

To address this limitation, eNetSTL utilizes a proxy-based method, introducing a proxy data structure to manage the ownership of all dynamically allocated memories. An overview of its functionality is provided below. Initially, upon each dynamic memory allocation, the ownership of the memory is transferred to the proxy for centralized management. Subsequently, we store the proxy in a BPF map, ensuring the persistence of all memories it manages. This method enables the persistent storage of a variable number of memories in the BPF map through the proxy data structure, overcoming the previously mentioned limitations. Additionally, we facilitate non-contiguous memory access by establishing relationships between two memories. These relationships are created using the `connect`[6] function (e.g., `A->next=B`) and released using the `disconnect` function (e.g., `A->next=NULL`). Assuming a relationship exists between two memories, the lightweight `get_next`[7] function allows us to access the subsequent memory based on the preceding one (e.g., `B=A->next`).

Although promising, such method introduces a new safety issue of use-after-free. The reason is that our method decouples the relationships between memories and their ownership, which results in the problem that releasing one memory does not lead to the release of the connected ones, as ownership belongs to the proxy data structure. For example, an eBPF-based network functions using two dynamic memories, A and B, that are connected (i.e., `A->next=B`). In a specific operation, if we do not disconnect B's relationship with A (i.e., `A->next=NULL`) before releasing B, a subsequent operation that directly accesses the next memory of A (i.e., `*(A->next)`) may trigger a use-after-free issue, ultimately leading to a kernel crash.

Lazy safety checking. To avoid the use-after-free issue, a straightforward approach involves checking the validity of the relationship before each memory access using `get_next`. Such validation can be accomplished by maintaining a hash table that stores all valid relationships. However, this method entails performing costly safety checks with every `get_next`, leading to a notable performance overhead.

Therefore, eNetSTL optimizes overall performance by delaying safety checks until the memory release, eliminating the need for safety checks in `get_next`. This approach is based on our observation that traversing non-contiguous

[5]This necessitates the allocation of dynamic memory using `bpf_obj_new` and its direct persistence into a BPF map. While eBPF linked list or rb-tree [16] can accommodate varying amounts of dynamic memory, they do not support pointer routing.

[6]The connect wrapper is necessary, as the eBPF verifier doesn't allow direct writing of memory returned from kernel functions

[7]The get_next wrapper is necessary to pass the eBPF verifier

```c
void bpf_mm256_mul_epu32 (u32 *dest,
  const u32 *lhs, const u32 *rhs) {
  /* Costly load: when using SIMD instructions, the operands
   * must first be loaded into the SIMD registers.
   * Abstracting each instruction as an API would mean
   * that every operation requires a memory read,
   * resulting in considerable overhead */
  __m256i lhs_vec = _mm256_loadu_si256(lhs);
  __m256i rhs_vec = _mm256_loadu_si256(rhs);

  /* Execute instruction*/
  __m256i dest_vec = _mm256_mul_epu32(lhs_vec, rhs_vec);

  /* Costly store: likewise, once the instruction is executed,
   * the result must be written back to memory from SIMD registers */
  __mm256_storeu_si256(dest, dest_vec);
}

int find_simd(const u32 *arr, u32 key) {
  /* Load only once: At the start of the algorithm,
   * we perform a single operation to load the input
   * from memory into the SIMD registers. */
  __m256i arr_vec = _mm256_loadu_si256(arr);

  /* Implement Alg using SIMD instructions */
  ...

  /* The result of the algorithm (the index of the matching element)
   * can be directly returned via the R0 register.*/
  return index;
}
```

Listing 1. Interfaces for parallel comparing

```c
void fasthash_simd(const void *key,
  size_t size, u32 *dest) {
  /* Use SIMD instructions to compute multiple fast hashes
   * for the same key, with the results temporarily
   * stored in SIMD registers */
  __m256i dest_vec = __fasthash_simd(buf, size);

  /* Costly store: Store the results from the SIMD registers
   * into the dest memory provided by the eBPF program */
  _mm256_storeu_si256(dest, dest_vec);
}

void hash_simd_cnt(void *buf, size_t buf_sz,
  const void *key, size_t ksize, u64 flags) {
  u32 row_sz = GET_ROW_SIZE(flags);
  u32 col_sz = GET_COL_SIZE(flags);
  /* Using SIMD to compute multiple hashes
   * with results stored in SIMD registers */
  __m256i dest_vec = __hash_simd(buf, size, flags);

  /* Post-hashing operations: retrieve each hash computation result
   * directly from the SIMD registers and increment the
   * corresponding counters based on these results. This method
   * eliminates the need to copy all hash results to memory before
   * performing the computation. */
  for (int i = 0; i < HASH_FN_NUM; i++) {
    /* Retrieve result from SIMD register */
    ++((u32*)buf + row_sz * i)
      [*((u32*)&dest_vec + i) & (col_sz)];
  }
}
```

Listing 2. Interfaces for multiple hash computing

memory (via `get_next`) is more frequent than connecting and releasing memories in network functions. For example, in a queueing operation [24] using a red-black tree with n nodes, the lookup operation relies solely on `get_next`, while the deletion operation involves $log(n)$ `get_next` invocations, along with a constant number of `connect` and a final release. In practice, eNetSTL records memory relationships in the `connect` function and automatically deletes all relationships associated with it based on the recorded information upon memory release, eliminating the need for safety checks in every `get_next` invocation. For example, consider two nodes, a and b, where `a->next = b`. When b is freed, `a->next` is automatically set to NULL based on the recorded information. This ensures that `a->next` is always either NULL or a valid pointer. Lazy safety checking prevents use-after-free issues in incorrect eBPF network functions. Well-implemented functions have updated memory relationships at the time of release, reducing the overhead of the release operation. The effectiveness of this design is demonstrated in §6.2.

4.3 Algorithms and Data Structures

Diving into shared behaviors, we further observe that in network scenarios, there is no need to employ a low-level interface for algorithms or data structures to achieve unnecessary generality. Based on this insight, eNetSTL reduces the interaction overhead without sacrificing necessary generality by adopting high-level interfaces, which is desired in the networking area. Table 2 summarizes the APIs provided by eNetSTL for the following algorithms or data structures:

Algorithms: bit manipulation. For operations using bit manipulation instructions, eNetSTL directly encapsulates individual bit instructions as interfaces. Despite being a low-level interface, this approach has a negligible impact on performance. The rationale behind this is that the input for these bit algorithms is a 64-bit bitmap, and the output is a numerical value, such as the index of the first set bit. The algorithms can efficiently utilize registers for input and output, eliminating the need for additional memory copies.

Algorithms: parallel comparing and reducing. For operations involving multiple buckets, optimization can be achieved by enabling parallel comparison and reduction using SIMD. The key idea here is that: instead of exposing low-level SIMD instructions directly, eNetSTL employs high-level interfaces to encapsulate SIMD instructions, delivering superb performance while satisfying all use cases for SIMD-based comparison and reduction. We avoid adopting a low-level interface due to the significant interaction overhead caused by memory loads and stores. To elaborate, before utilizing SIMD instructions, data must be loaded from memory into SIMD registers[8] (referred to as SIMD loads). Upon completion of SIMD instructions, the results are cached in another SIMD register and can be reloaded back into memory (referred to as SIMD stores). Thus, each invocation of the low-level interface involves loading data from eBPF memory into registers and copying the result back to eBPF upon completion. For example, parallel comparison and reduction require at least two SIMD instructions to complete. Thus, using low-level interfaces introduces significant performance overhead. As shown in Listing 1, we provide an example of

[8]SIMD instructions depend on specific registers, e.g., 256-bit registers for AVX256

eNetSTL	Observations	APIs	Description	Involved works
Wrapper	Shared behavior 5	`node_alloc(release)` `(un)set_owner` `node_(dis)connect` `get_next,` `node_write`	Memory wrapper which adopts a proxy-based memory ownership management and and lazy safety checking	NFD-HCS [47], FQ/pacing [24], Space Saving [50]
Algorithms	Shared behavior 1	`ffs` (↑ 1.5×) `popcnt` (↑ 1.5×)	Corresponding bit manipulation instruction	Rand-indexed [34], VBF [36], Tiny Table [22], UnivMon [46], Eiffel [64], Non-Cascade TW [72]
	Shared behavior 6	`find_simd` (↑ 2.1×) `reduce_simd` (↑ 2.1×)	Parallel comparison and reducing max/min algorithms based on SIMD	Cuckoo Switch [59], CSS [5], HSS [55], Tiny Table [22], Cuckoo Filter [25], SILT [44], d-left BF [10], Blocked BF [61], RHSS [6], HeavyKeeper [81]
	Shared behavior 2	`hw_hash_crc` (↑ 1.5×) `hash_simd_cnt` (↑ 5.7×) `hash_simd_bit` (↑ 5.7×) `hash_simd_comp` (↑ 5.7×)	Computing multiple hash functions based on SIMD and take actions on memory	d-ary Cuckoo [27], Bloom Filter [8], Summary cache [26], HeavyKeeper [81], Count Min [15], UnivMon [46], Sketch Visor [35], Elastic Sketch [80], Nitro Sketch [45], EFD [20]
Data Structures	Shared behavior 3	`list_buckets` (↑ 1.7×)	Data structure of List-buckets implemented in eNetSTL	Eiffel [64], PCQ [66], Carousel [75], Non-Cascade TW [72]
	Shared behavior 4	`random_pool` (↑ 6.1×)	Random number pools with auto-reinjection	RHSS [6], Memento [3], Nitro Sketch [45]

Table 2. Summary of the wrapper, algorithms, and data structures in eNetSTL. The third column illustrates the performance enhancement of them over the eBPF implementations. (indicated by ↑, with numerical values representing the performance improvement ratio)

`bpf_mm256_mul_epu32` (lines 1-17), which wraps the SIMD `mm256_mul_epu32` instruction. Each invocation incurs costly loads (lines 8-9) and expensive stores (line 16).

Instead of using the aforementioned low-level instruction-level exposure, eNetSTL directly provides a high-level interface of parallel comparison and reduction algorithms. Because the high-level one reduces the number of memory copies, the two algorithms are already sufficient for operations involving operations on multiple buckets. The algorithms take the input memory and key as input, while directly returning a small result, such as the index of the first matched item, which avoids additional memory copies introduced by storing intermediate results. As shown in Listing 1, we provide a simplified version of parallel comparing algorithm `find_simd`. It involves only one input load (line 23) and returns the final result directly (line 30).

Algorithms: unified post-hashing operations. The calculation of a single hash function can take advantage of hardware-based CRC instructions, such as practices [36] in DPDK. Consequently, the computation of a single hash function can be encapsulated within the `hw_hash_crc` algorithm.

For operations involving multiple hash calculations, they can be optimized with SIMD. However, utilizing a low-level interface that takes a key as input, performs calculations with multiple hash functions, and copies the results back to output memory introduces significant overhead. This issue arises because the results of multiple hash functions are too large to be directly returned via registers, and eBPF cannot directly access the results temporarily stored in SIMD registers. The fundamental problem lies in the absence of SIMD instruction support within the eBPF instruction set. Consequently, two extra memory copies are required to retrieve the calculation results: first from SIMD registers to eBPF memory and then from eBPF memory to eBPF registers. Overcoming this constraint would necessitate expanding the eBPF instruction set, introducing substantial intrusion into the kernel. As illustrated in Listing 2, we provide a counterexample of *fasthash_simd* (lines 1-11), which computes multiple fasthash on a single key. The `store` instruction (line 10) at the end of the algorithm negates the performance improvement brought by SIMD (line 6).

Therefore, eNetSTL provides high-level interfaces that encapsulate multiple hash calculations and the subsequent operations within a single algorithm. This method is feasible with our observation that NFs usually rely more on how to utilize the results of multiple hash calculations than the values themselves. For example, NFs utilize the results of multiple hash calculations to determine memory positions and execute subsequent operations at these specific locations (e.g., incrementing counters). These post-hash operations do not necessitate returning results, or the size of results is much smaller than that of multiple hashing values. Therefore, eNetSTL unifies hashing and post hashing operations by offering algorithms such as counting after hashing [15, 26, 35, 45, 46, 80], setting bits after hashing [8], and comparing after hashing [27]. This approach eliminates the need for intensive memory copying, ensuring high performance of NFs. As illustrated in Listing 2, we present the algorithm *hash_cnt_simd* (lines 13-31), which computes hash functions in parallel (line 19) and then increments counters based on hash results (lines 26-30). The post-hash operation, i.e., counting, avoids transferring the original hashing values and consequential expensive memory copies.

Data structure: list-buckets. For NFs that leverage linked lists to store data elements, eNetSTL provides a high-level data structure called list-buckets instead of directly utilizing the linked list data structure provided by eBPF. This design is driven by two root causes of decreased performance: (1) eBPF mandates that each insertion and deletion operation on a linked list must contend for locks, which is performance costly; (2) Using an array of linked lists for NFs requires multiple BPF map elements to store each list separately, leading to extra eBPF helper function calls.

eNetSTL provides a high-level data structure of list-buckets (i.e., bucket-queues) based on our observation that NFs almost utilize multiple linked lists simultaneously. Therefore, our design of list-buckets offers a unified API that uses parameters to select the target queue for insertion. It helps avoid frequent calls to `bpf_map_lookup_elem` to access the underlying singly linked list (queue). List-buckets further avoid lock contention by holding percpu data, thereby enhancing the performance.

Data structures: random-pool. Generating random numbers is costly in eBPF. For operations utilizing random updating, we borrow the idea from previous works [52], abstracting the random number pools as a shared data structure. Furthermore, we enhance the previous work, which only use a fixed pool, by designing a mechanism that can automatically reinject random numbers. eNetSTL avoids rapid pool depletion by adopting a solution from prior research [45, 52], using a specialized *geo_rpool* to generate random numbers following a geometric distribution.

4.4 Safety of eNetSTL

In general, eBPF provides several safety guarantees, including safe program termination, memory safety, limited program complexity, access control to kernel data structures, and runtime privilege checks. Since eNetSTL is a self-contained library and not designed for interaction with the kernel, we primarily focus on two primary safety properties . Firstly, eBPF guarantees that programs can safely terminate. This implies the absence of unbounded loops, elimination of out-of-bound jumps, and prevention of runtime exceptions (e.g. division by zero error). Secondly, it ensures memory safety by avoiding out-of-bound memory access, preventing resource leaks, and avoiding use-after-free issues.

eNetSTL initially implements these APIs in Rust, supplemented by manual audits of unsafe code to enhance the inherent safety of each API. While ensuring the safety of individual APIs guarantees the safe termination of the entire eBPF program, it does not fully address memory safety due to the differing memory models of eBPF and Rust. Within eBPF programs, we cannot depend on Rust's memory model, such as RAII, to protect the memory of resources returned by APIs. For instance, eBPF programs must call alloc APIs and release APIs in pairs to release resources.

eNetSTL further ensures the safe interaction between eBPF and eNetSTL for memory safety. Leveraging Rust and code reviewing , eNetSTL now offers a collection of memory-safe APIs. Thus, by ensuring that eBPF programs adhere to correct API usage—such as explicitly calling release APIs for all alloc APIs, validating input parameters, and preventing out-of-bounds access to API-returned memory—we can uphold the memory safety of the eBPF program. eNetSTL accomplishes this by incorporating metadata into APIs (i.e., add annotations for kfuncs), providing guidance to the eBPF verifier, which enforces the correct usage of these APIs in eBPF programs. We now outline the technical details of eNetSTL that contribute to safe termination and memory safety.

Avoid runtime exceptions and out-of-bound jump: We utilize open-source rust-no-panic macro [21] to ensure, during compile time, that our code remains free from panics. This requires us to explicitly incorporate boundary check codes in areas where implicit calls to `panic!` could arise, such as during division operations and array indexing. As for out-of-bound jumps, Rust's language features ensure they do not occur in APIs implemented in Rust [38].

Avoid out-of-bound memory access: For API safety, we introduce a lightweight safe abstraction layer by leveraging unsafe Rust to convert raw pointers into types for safe Rust. Specifically, converting pointers into references to their corresponding types enables safe Rust to access memory from eBPF programs without worrying about out-of-bounds issues. For the safe interaction, We provide the function signature metadata of the API to the verifier, such as whether the parameters are stack memory or if the return value might be a null pointer, etc., and the verifier enforces that eBPF programs pass valid arguments to APIs and access memory returned from APIs in-bound.

Avoid memory leaks and use-after-free: For API safety, we implement a safe abstraction layer using smart pointers with reference counting to manage heap memory, including memory allocated by kernel APIs, input arguments of release APIs, and internal memory of data structures (e.g., linked list nodes). At the abstraction layer just before the API exit, we convert smart pointers to raw pointers, transferring memory ownership to the eBPF program. This approach ensures that all heap memory is either properly released or its ownership is transferred back to the eBPF program, while also guaranteeing that all addresses returned by the APIs are safeguarded by reference counting. To ensure safe interaction, we annotate the alloc API and APIs that increment reference counts with KF_ALLOC. Conversely, the release API and APIs that decrement reference counts are annotated with KF_RELEASE. The eBPF verifier confirms that these APIs are used in pairs.

Limitations and manually reviewing: Although implementing eNetSTL in Rust effectively mitigates safety risks, we cannot ensure its safety through Rust alone without code reviews. This limitation arises from two reasons. Firstly, eNetSTL uses unsafe Rust in its implementation. While safe Rust guarantees memory safety through its ownership and borrow checker and by prohibiting raw pointers, unsafe Rust is required to enable interactions between the kernel's eBPF program and safe Rust. Furthermore, certain APIs in eNetSTL need to use low-level instructions, like SIMD, which are also encapsulated in unsafe Rust. Secondly, safe Rust cannot

```
1  typeof struct ptr_list_node {
2    void* outs[1];
3    void* ins[1];
4    OTHER_META_DATA;
5    char data[DATA_SIZE];
6  } list_node;
7
8  void list_add(struct node_list *nl,
9    list_node *head, void *data) {
10   /* Alloc new node with one out ptr and one in ptr */
11   list_node *new_entry = node_alloc(1, 1);
12   if (new_entry == NULL)
13     return; /* Necessary to pass the verifier */
14   set_owner(nl, new_entry);  /* Proxy-based memory management */
15   /* head->out[0] */
16   list_node *next = get_next(head, 0);
17   if (next == NULL) {
18     /* head->out[0]=new_entry;
19      * new_entry->in[0]=head; */
20     node_connect(head, 0, new_entry, 0);
21   } else {
22     node_connect(new_entry, 0, next, 0);
23     node_connect(head, 0, new_entry, 0);
24     node_release(next);
25   }
26   /* Required by verifier */
27   node_write(new_entry, OFF, data, DATA_SIZE);
28   /* The node will not be free because of set_owner */
29   node_release(new_entry);
30 }
```

Listing 3. Example of Case Study 1

fully ensure safe termination because the current toolchain cannot verify that unbounded loops will safely exit.

Currently, we address this limitation through manual code reviews. Additionally, the current implementation of eNetSTL does not use unbounded loops. We believe that as research on Rust and the kernel advances [30, 38], this limitation can be overcome. For example, eBPF kfunc and kptr mechanisms could provide Rust interfaces, and the kernel could offer more comprehensive Rust components, such as new data structures and memory allocators. This would enable the development of in-kernel libraries entirely in safe Rust. Additionally, runtime checks can be implemented to ensure that unbounded loops eventually terminate [38].

4.5 eNetSTL for Future NFs

Currently, eNetSTL has already fully covered the 35 research works. Furthermore, we believe that eNetSTL can benefit future works for two reasons. Firstly, we believe that future works will to a certain extent follow the designs from these representative works, making eNetSTL directly applicable to them. For instance, computing multiple hash for the same key is likely to remain a common pattern of sketching. Moreover, the incorporation of support for non-contiguous memory significantly enhances eBPF's flexibility in facilitating other NFs, such as LRU based on lists. Secondly, even if future works adopt new designs, which may not be directly covered by eNetSTL now, they can still benefit from the idea provided by eNetSTL and extract new components from their designs, which can be further added to eNetSTL. Although this involves upgrading the in-kernel eNetSTL library, we believe such upgrading is less frequent and tends to become rare while eNetSTL becomes more mature.

5 Case Study

In this section, we use case studies to demonstrate how eNetSTL can solve the problems of incomplete functionalities and performance degradation. Due to space limitations, we only select three illustrative cases here, which use the memory wrapper, algorithms, and data structures, respectively.

5.1 Case Study 1: Key-value Query in NFD-HCS

NFD-HCS [47] employs a key-value query operation based on skip-list within its in-network caching system. Enabled by the memory wrapper of eNetSTL, the implementation of skip-list is now feasible in eBPF, owing to the support for non-contiguous memory. We use the `add_node` operation of link-list as code examples; the principles of skip-list operations remain consistent.

The crucial aspect of utilizing the memory wrapper is to follow these steps: (1) Allocate a new node using `node_alloc` and delegate the management of the node's ownership to the proxy structure using `set_owner`. (2) Invoke `node_release` to destroy a node from the proxy data structure. And this function will update relationships between nodes as mentioned before. (3) The `node_(dis)connect` function facilitates the (dis)connection of two nodes (4) Invoke `get_next` to obtain a reference to the next node and use `node_release` to release the reference. Reference counting is employed in `get_node` to prevent use-after-free issues and the node will not be actually freed until invoking `unset_owner` to detach itself from proxy-based memory management.

Listing 3 gives an example of implementing `list_add` interface of linked list using eNetSTL memory wrapper. To add a new node, it first allocates a new node and leverages the proxy structure `node_list` to manage the ownership (lines 10-14). Then it gets the reference of the next node of the head (i.e., `head->next`) (line 16). It should be noted that this step increases the reference count and involves zero safety checks. Finally, it reconnects nodes (lines 20, 22-23) and releases references (lines 24, 29).

5.2 Case Study 2: Count-min sketching

Count-min sketch [15] is a widely used sketching operation. We illustrate how to leverage the algorithm provided by eNetSTL to optimize its implementation. Invoking an algorithm in eNetSTL is similar to regular function calls. As Listing 4 shows, the steps to implement Count-min sketch (CM) are as follows: (1) we define the memory layout of CM (lines 1-3). (2) we define a BPF map instance to handle the underlying memory of CM (line 6). (3) when the number of hash functions is small we leverage a `hw_hash_crc` algorithm (line 22), otherwise, we use the parallel hash after counting algorithm `hash_simd_cnt` (line 29) for optimization.

5.3 Case Study 3: Queuing in Carousel

The Time Wheel [75] constitutes a fundamental logic within Carousel [63]. Carousel relies on the time wheel to queue

```
struct cm_sketch {
  u32 cm[ROW_SZ][COL_SZ];
};

/* Use BPF MAP to handle data structure memory */
BPF_PERCPU_ARRAY(map, struct cm_sketch, 1);

/* Implementing data structure operation using eBPF */
static void ebpf_countmin_add(struct cm_sketch *cmk, void *pkt) {
  /*multiple hash computing*/
   for (int i = 0; i < COL_SZ: i++) {
        /* Utilize a software-based implementation
         * of xxhash to compute each hash function sequentially */
        u32 hash = sw_xx_hash();
        ++cmk->cm[i][hash & COL_SZ];
   }
}
/* Implementing data structure operation using eNetSTL */
static void enetstl_coutmin_add(struct cm_sketch *cmk, void *pkt) {
#if COL_SZ <= 2
    for (int i = 0; i < COL_SZ; i++) {
      u32 hash = hw_hash_crc(pkt, seeds[i]);
      ++cmk->cm[i][hash & COL_SZ];
    }
#else
    /* Use SIMD to accelerate the computation
     * of multiple hash functions */
    hash_simd_cnt(&cmk->cm, SIZE, pkt, PKT_SIZE,
      make_flags(ROW_SZ, COL_SZ, HASH_NUM));
#endif
}
```

Listing 4. Example of Case Study 2

packets based on their transmission timestamps. In this part, we introduce how to implement a time wheel using eNetSTL, which uses the list-buckets data structure provided by eNetSTL within eBPF.

To utilize the data structures in eNetSTL, based on kptr [17], the main steps are as follows. Firstly, the `alloc` function of the data structure is invoked to create and initiate an instance (kptr) of the data structure. Secondly, the instance is stored in a BPF map for later use. Thirdly, the data structure can be intentionally destroyed by invoking the `destroy` function, or it can be automatically destroyed along with the BPF map. Finally, operations of the data structure are invoked using the instance as a parameter.

Here, we provide an example of a one-level time wheel (i.e., calendar queue) in Listing 5. This data structure uses the list-buckets in eNetSTL and keeps an instance of it in BPF map (line 3). On the procedure of add (lines 8-20), it first retrieves or creates the instance of the bucket lists (line 10). Then it invokes the `insert` (line 18) function to insert data into the bucket calculated by the `expires`.

6 Evaluation

In this section, we first present our evaluation methodology. Then, we show the effectiveness of eNetSTL by evaluating the aforementioned three case studies, and further including 8 extra cases to fully cover all categories of NFs shown in Table 1.

6.1 Methodology

Environment Setup: Our testbed includes two servers connected back-to-back with a dual-port Intel XL710 40Gbps

```
1  struct time_wheel {
2    u64 clk;
3    struct bucket_list __kptr *bktlist_lvl_1;
4  }
5
6  BPF_PERCPU_ARRAY(map, struct time_wheel, 1);
7
8  static void add(struct time_wheel *tw,
9    void *data, u64 expires) {
10   struct bucket_list *bl = get_or_init(tw);
11   if (bl == NULL) return;
12   unsigned long idx = expires - base->clk;
13   /* Calculate the index of the bucket */
14   if (idx < TVR_SIZE) expires =  tw->clk + TVR_SIZE - 1;
15   /* Calculate the index of bucket */
16   int i = expires & TVR_MASK;
17   /* Insert into the i-th queue using the unify API */
18   bktlist_insert_front(bl, i, data, DATA_SIZE);
19   set_or_release(bl);
20 }
```

Listing 5. Example of Case Study 3

NIC. Both servers are equipped with two Intel(R) Xeon(R) CPU E5-2630 v4 @2.20GHz (20 cores), 128GB memory, and with Linux kernel v6.6. One server acts as a traffic sender and the other one acts as a receiver where eBPF programs are attached to the XDP hook in native mode. The evaluation setup is similar to the recent work [54].

Traffic and performance metric: we use pktgen [37] with DPDK 22.11 to replay packet traces consisting of randomly generated 64-byte packets and report results. For certain NFs, their performance is impacted by the load and configuration. For example, the performance of the key-value query is influenced by the load factor (i.e., the ratio of the current number of elements to the maximum capacity) of the underlying table, and the performance of the sketching is sensitive to the number of hash functions employed. Thus, we evaluate the performance by varying the factors that have the greatest impact on their performance. Detailed testing methods are explained in each evaluation. We report single-core metrics, including throughput and latency, by configuring the receiver-side scaling (RSS) to redirect all flows to a single queue bonded to a single CPU on the same NUMA node. For throughput, the packets are directly dropped after being processed by the NF, and we report the packet-per-second (PPS) rate. For latency, after processing, the NF forwards the packets back to the sender, and the end-to-end latency is measured by subtracting the send timestamp from the receive timestamp. For each evaluation, we conduct five trials, report the average, and include error bars to represent variability. The setup is similar to the existing works [52, 54]

6.2 Network Functions

In this part, we illustrate how eNetSTL can address the two problems of incomplete functionality and performance degradation by evaluating 11 representative network functions, with their core components implemented with pure-eBPF (eBPF), in-kernel (Kernel), and eNetSTL (eNetSTL).

Case Study 1: Key-value query in NFD-HCS [47]. We implement skip-list-based key-value query in NFD-CHS based

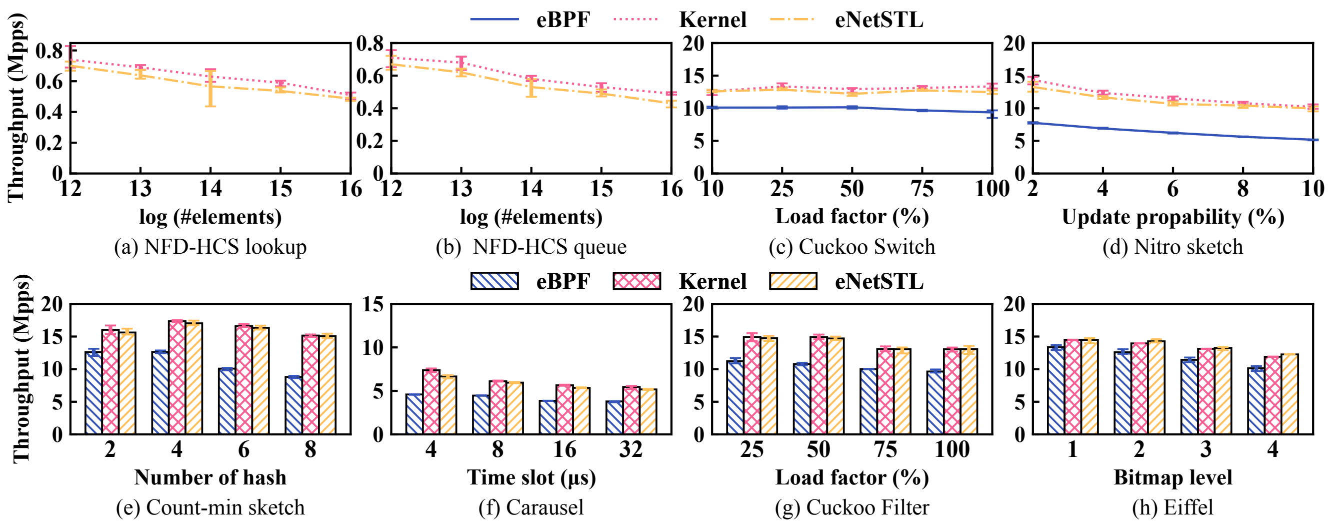

Figure 3. Performance of NFs under various configurations.

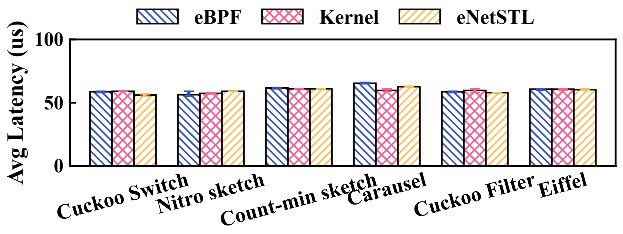

Figure 4. Latency of NFs under low load (1kpps).

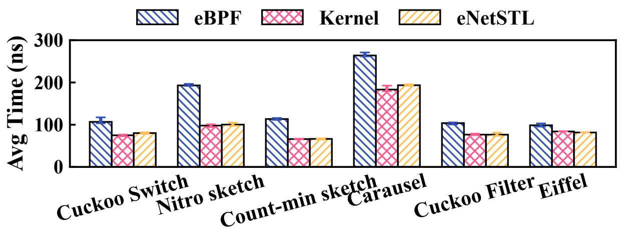

Figure 5. Per-packet processing time of NFs.

on the memory wrapper of eNetSTL. The skip-list is set with a maximum height of 16. The key size and the value size are 32B and 128B, respectively. It should be noted that the key-value query based on skip list is not feasible in native eBPF as we have already shown in §2.2, so we compared eNetSTL with the kernel. We conduct two evaluations. Firstly we evaluate the lookup operation, which finds elements with a specific key parsed from the packet. Secondly, we evaluate the performance of update and delete, where update packets and delete packets arrive in a ratio of 1:1. For both tests, we evaluate the performance under different loads (i.e., the number of elements in the current key-value map). The results are shown in Figure 3(a) and (b). In the case of the lookup operations, there is a small performance gap of around 7.33% between eBPF and the kernel. This is because, despite our efforts to eliminate safety checks in the `get_next` operation, some inherent safety mechanisms of eBPF, such as checking if the pointer returned by kernel function is empty, cannot be eliminated. However, considering that eNetSTL addresses the problem of incomplete functionality in eBPF-based NFs, this performance gap is acceptable. For updates and deletes, the results are similar to lookup with an average gap of 8.54%. The reason is although eNetSTL's lazy safety checking strategy adds overhead in memory release and connection, the get_next operation remains the primary operation.

Case Study 2: Count-min sketching [15]. We use `hw_hash_crc` and `hash_simd_cnt` in eNetSTL to implement Count-min sketch using xxhash as the hash function. The performance of a sketching operation is mainly impacted by the number of hash functions it employs. Hence, we evaluate sketches with varying numbers of hash functions. The results depicted in Figure 3(e) illustrate that eNetSTL achieves an average performance improvement of 47.9% compared to eBPF. Notably, the enhancement becomes more pronounced with an increasing number of hash functions, peaking at 70.9% with 8 hash functions. This is due to the improved optimization effect achieved by SIMD instruction as the number of hash functions increases. When the number of hash functions is low (<=2), we replace them with hw_hash, serving as a single hash function. The performance of eNetSTL and kernel is nearly identical with an average gap of 1.64%.

Case Study 3: Queueing in Carousel [63]. Carousel uses the time wheel to queue packets by sending timestamps. We implement queuing based on the two-level time wheel in this case and evaluate the performance of enqueue and dequeue operations at various time slot granularity. Figure 3(f) shows that eNetSTL outperforms eBPF by an average of 38.4% due to the more efficient list-buckets data structure in eNetSTL compared to eBPF's native linked list. On average, eNetSTL's performance lags behind the kernel by 5.75%.

Cuckoo Switch [82]. Cuckoo Switch involves the core component of key-value query based on Blocked Cuckoo hash [19] and we implement it using `hw_hash_crc` and `find-_simd` in eNetSTL. The performance is primarily dependent on the current load of the Blocked Cuckoo hash (i.e., the average number of elements in each entry). We conduct evaluations on the performance at various loads, using the packet 5-tuple as the key and 8-byte dest port as the value. As illustrated in Figure 3(c). eNetSTL achieves an average performance improvement of 27.4% compared to eBPF, with a more pronounced enhancement as the load increases, reaching up to 33.08% at full load. This is because, as the load grows, the average number of comparisons in a single entry rises. Enhanced optimization is achieved through SIMD-based parallel comparisons. In low-load scenarios, the optimization primarily entails using `hw_hash_crc` as a replacement for software-based hash calculations and the SIMD-optimized full-key comparisons. Compared to the kernel, eNetSTL exhibits an average performance loss of approximately 4.30%.

Nitro sketch [45]. We use `random_pool` and `hw_hash_crc` in eNetSTL to implement it. In the eBPF version, we use `bpf_get_prandom_u32` to generate a random number for each row to determine whether to update that row. In a low update probability, random number generation plays a dominant role, while a high update probability shifts the focus to hash calculation. Therefore, we set the number of rows to 8 and conduct evaluations under different update probabilities. As shown in Figure 3(d). Leveraging both a random number pool and hardware hash optimization, eNetSTL achieves an average performance boost of 75.4% compared with eBPF and an average gap of 5.24% compared with the kernel.

Cuckoo Filter [25]. Cuckoo Filter employs a hash scheme to perform membership test operations. We implement it using `hw_hash_crc` and `find_simd` in eNetSTL. We evaluate it at various loads, using the packet 5-tuple as the key and testing whether a flow belongs to a set. The results, as depicted in Figure 3(g), show that as the load increases, the optimization effect of eNetSTL becomes more pronounced. On average, it achieves a 31.8% optimization, with a 35.7% improvement under full load. Compared to the kernel, eNetSTL exhibits a limited performance loss of 0.8%.

Eiffel [64]. Eiffel utilizes cFFS, a bitmap-based priority queue, for packet queueing. We implement it using `ffs` in eNetSTL. The performance is primarily on the priority range, i.e., the number of FFS (find the first set bit) queries on the bitmap. We evaluate the enqueuing and dequeuing performance of cFFS at different levels which represents a 64^{level} maximum distinct priorities. Figure 3(h) indicates that, compared to eBPF, eNetSTL achieves an average performance improvement of 14.6%, with a more pronounced enhancement as the level increases, reaching up to 20.90% at level 4.

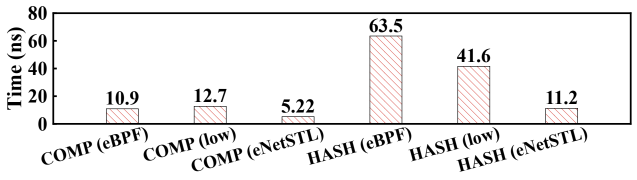

Figure 6. Performance of two observed behaviors: arranging multiple bucket (COMP) and using multiple hash (HASH). Low means using low-level interfaces.

Additionally, The performance of eNetSTL is nearly identical to that of the kernel.

Other cases. We also implemented the following NFs using eNetSTL: load balancing of EDF [20], packet classification of TSS [68], counting of HeavyKeeper [81], and membership testing of VBF [36]. Compared to eBPF, the average performance improvements are 48.3%, 26.7%, 30.0%, and 15.8%, respectively. And compared to the kernel implementation, eNetSTL exhibits an average performance gap of 4.71%, 3.96%, 2.53%, and 2.62%, respectively.

Summary. eNetSTL can extensively optimize various NFs, showing more pronounced improvements, especially in scenarios characterized by heavy traffic loads or configurations. This is because, in heavy-load scenarios, the proportion of performance contributed by the eBPF codes replaced by eNetSTL increases, resulting in better optimization. This highlights that eNetSTL, while enhancing overall performance, is particularly well-suited for worst-case scenarios or situations where leveraging more complex configurations for achieving better algorithmic metrics (e.g., the accuracy of sketches) without compromising performance. Furthermore, compared to the kernel, the performance loss introduced by the encapsulation in eNetSTL is within an acceptable range considering its flexibility and generality.

6.3 Latency Evaluations

In this part, we evaluate different types of network functions, measuring latency when implemented using eBPF, LKM, and eNetSTL. For each NF, we conduct tests under heavy configurations. For example, for the cuckoo-switch, we measured latency at a 100% load factor. Figure 4 shows the results (using a sending rate of 1k pps). We observe that compared to eBPF, eNetSTL does not significantly increase latency. We further measure the per-packet processing time by invoking `bpf_ktime_get_ns` at the start of the program and just before returning. As shown in Figure 5, eNetSTL reduces per packet processing time compared to a pure eBPF implementation. The experiment results demonstrate that eNetSTL does not increase the end-to-end latency of network functions. This is because eNetSTL speeds up the processing of individual packets in eBPF programs without introducing mechanisms like batching that could increase the delay of packets received earlier.

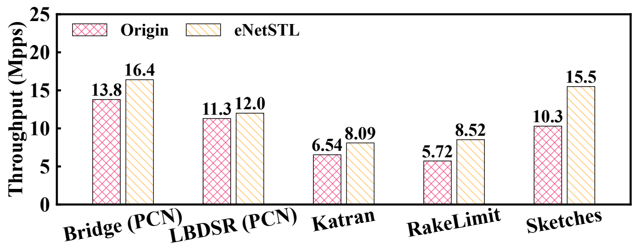

Figure 7. Performance of integrating eNetSTL into a real-world eBPF project (eNetSTL) and original version (Origin)

6.4 Performance Breakdown

In this part, we evaluate the algorithms and data structures listed in Table 2. The partial results, highlighting the performance improvement realized by eNetSTL in comparison to pure eBPF, are directly showcased in the third column of the table (indicated by ↑, with numerical values representing the performance ratio). It should be noted that the reported performance in this part is the average time required for individual operations, excluding packet parsing. The overall performance improvement of a single algorithm or data structure reaches 52.0%-513%. Furthermore, we demonstrate the significance of rational abstraction by evaluating the performance of different interfaces discussed in §4.3. The results are depicted in Figure 6. In comparison to the current implementation of eNetSTL, these methods show a performance degradation of 59.0% to 73.1%.

6.5 eNetSTL in Action

In this part, we integrate eNetSTL into real-world eBPF projects including PolyCube (PCN) [53], Katran [57], RakeLimit [39], and open-source eBPF-based sketches [52] and improve their performance, demonstrating that eNetSTL is a practical solution. We replace the core components (based on BPF maps) used in these applications with ones implemented based on eNetSTL. For instance, we replace the key-value queue operation based on the BPF hash with the one based on the Blocked Cuckoo hash and the Count-min sketch in RakeLimit with the eNetSTL version. As shown in Figure 7, eNetSTL improves the performance by 21.6% on average.

7 Related Work

Software network function: Software network functions are crucial components of modern network infrastructure. They can be classified into seven widely used categories: key-value query [27, 44, 47, 59, 82], membership test [8, 10, 25, 26, 34, 36, 61], packet classification [32, 67, 74], load balancing [20, 23, 58], counting [3, 5, 6, 22, 50, 55, 81], sketching [15, 35, 45, 46, 80], and queuing [24, 63, 64, 66, 72].

eBPF for networking: eBPF has become a promising option for implementing network functions. Recent work includes fundamental frameworks for data planes such as Cilium [14], SPRICHT [62], and PolyCube [53]. It also encompasses network measurement services based on eBPF, including ViperProbe [42], Nethint [13], and eBPF-sketch [52]. Protocol stack optimization includes eMPTCP [78] and PQUIC [18], and network optimization for specific applications such as BMC [29], and Electrode [83] were also studied. In addition, hXDP [9, 12] offloads XDP to FPGA for acceleration.

Improve the performance of eBPF programs: Optimizing the performance of general eBPF programs can be achieved by code optimization. For example, K2 [77] utilizes program synthesis techniques to generate a more efficient version of eBPF bytecode that still passes the verifier. J. Mao et al. [48] improve performance through LLVM Instruction Representation (IR) transformation and bytecode refinement. H. Kuo et al. [41] observed performance degradation in eBPF applications due to the chain pattern of multiple eBPF programs and introduced KFuse to rewrite tail calls (indirect call) between verified eBPF programs as direct calls. In the network domain, Miano et al. [54] introduced Morpheus, which dynamically optimizes eBPF-based network functions with fast-path code by analyzing the locality of traffic. We believe that eNetSTL's approach complements these solutions. For instance, K2 could also be used to optimize eBPF programs utilizing eNetSTL.

Rust for operating system. Rust has become a popular programming language for OS [11, 30, 38, 43, 56]. Notable work includes building OS from scratch [11, 56], improving the safety of Linux [43], and implementing drivers [30] and kernel extensions using Rust [38]. Specifically, regarding eBPF, J. Jia et al. [38] pointed out that due to the unverified nature of helper functions and their increasing number, eBPF's verifier has become a liability. Thus they proposed using Rust for developing kernel extensions, combined with runtime checks to ensure safety. However, this approach of replacing the entire eBPF subsystem is still far from being widely adopted. eNetSTL builds on this idea by using Rust to develop in-kernel libraries while maintaining compatibility with the existing eBPF infrastructure.

8 Conclusion

In this paper, we identified problems with incomplete functionalities and performance degradation in 35 representative NFs using eBPF. We introduced eNetSTL, an in-kernel library for high-performance eBPF-based NFs. eNetSTL is designed based on the observation that NFs exhibit similar performance-critical behaviors, allowing us to abstract them into a minimal and stable set of components.

Acknowledgements

This work is supported by National Natural Science Foundation of China under Grant 62272101, 62402407, 61906040, 61972085, Natural Science Foundation of Jiangsu Province under grant BK20230083, Peking University startup funding.

References

[1] Zaafar Ahmed, Muhammad Hamad Alizai, and Affan A. Syed. 2018. InKeV: in-kernel distributed network virtualization for DCN. *SIGCOMM Comput. Commun. Rev.* 46, 3, Article 4 (jul 2018), 6 pages. https://doi.org/10.1145/3243157.3243161

[2] Alexei Starovoitov. Accessed September. 2024. Re: Supporting New Memory Barrier Types in BPF. https://lore.kernel.org/bpf/CAADnVQJqGzH+iT9M8ajT62H9+kAw1RXAdB42G3pvcLKPVmy8tg@mail.gmail.com/. (Accessed September. 2024).

[3] Ran Ben Basat, Gil Einziger, Isaac Keslassy, Ariel Orda, Shay Vargaftik, and Erez Waisbard. 2018. Memento: Making Sliding Windows Efficient for Heavy Hitters. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '18)*. Association for Computing Machinery, New York, NY, USA, 254–266. https://doi.org/10.1145/3281411.3281427

[4] bcc. Accessed Jan. 2024. BPF Features by Linux Kernel Version. https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md. (Accessed Jan. 2024).

[5] Ran Ben-Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. 2016. Heavy hitters in streams and sliding windows. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*. 1–9. https://doi.org/10.1109/INFOCOM.2016.7524364

[6] Ran Ben Basat, Gil Einziger, Roy Friedman, Marcelo C. Luizelli, and Erez Waisbard. 2017. Constant Time Updates in Hierarchical Heavy Hitters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 127–140. https://doi.org/10.1145/3098822.3098832

[7] Bill Mulligan, Daniel Borkmann. Accessed May. 2024. The Silent Platform Revolution: How eBPF Is Fundamentally Transforming Cloud-Native Platforms. https://www.infoq.com/articles/ebpf-cloud-native-platforms/. (Accessed May. 2024).

[8] Burton H. Bloom. 1970. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (jul 1970), 422–426. https://doi.org/10.1145/362686.362692

[9] Marco Bonola, Giacomo Belocchi, Angelo Tulumello, Marco Spaziani Brunella, Giuseppe Siracusano, Giuseppe Bianchi, and Roberto Bifulco. 2022. Faster Software Packet Processing on FPGA NICs with eBPF Program Warping. In *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, Jiri Schindler and Noa Zilberman (Eds.). USENIX Association, 987–1004. https://www.usenix.org/conference/atc22/presentation/bonola

[10] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. 2006. An improved construction for counting bloom filters. In *Algorithms–ESA 2006: 14th Annual European Symposium, Zurich, Switzerland, September 11-13, 2006. Proceedings 14*. Springer, 684–695.

[11] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. 2020. Theseus: an Experiment in Operating System Structure and State Management. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 1–19. https://www.usenix.org/conference/osdi20/presentation/boos

[12] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. 2020. hXDP: Efficient Software Packet Processing on FPGA NICs. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 973–990. https://www.usenix.org/conference/osdi20/presentation/brunella

[13] Jingrong Chen, Hong Zhang, Wei Zhang, Liang Luo, Jeffrey S. Chase, Ion Stoica, and Danyang Zhuo. 2022. NetHint: White-Box Networking for Multi-Tenant Data Centers. In *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022*, Amar Phanishayee and Vyas Sekar (Eds.). USENIX Association, 1327–1343. https://www.usenix.org/conference/nsdi22/presentation/chen-jingrong

[14] Cilium official website. Accessed Jan. 2024. Cilium: eBPF-based Networking, Observability, Security. https://cilium.io/. (Accessed Jan. 2024).

[15] Graham Cormode and S. Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75. https://doi.org/10.1016/j.jalgor.2003.12.001

[16] Dave Marchevsky. Accessed Jan. 2024. BPF rbtree next-gen datastructure. https://lwn.net/Articles/917201/. (Accessed Jan. 2024).

[17] David Vernet. Accessed Jan. 2024. Long-lived kernel pointers in BPF. https://lwn.net/Articles/900749/. (Accessed Jan. 2024).

[18] Quentin De Coninck, François Michel, Maxime Piraux, Florentin Rochet, Thomas Given-Wilson, Axel Legay, Olivier Pereira, and Olivier Bonaventure. 2019. Pluginizing QUIC. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 59–74. https://doi.org/10.1145/3341302.3342078

[19] Martin Dietzfelbinger and Christoph Weidling. 2007. Balanced allocation and dictionaries with tightly packed constant size bins. *Theoretical Computer Science* 380, 1-2 (2007), 47–68.

[20] DPDK. Accessed Jan. 2024. Elastic Flow Distributor Library. https://doc.dpdk.org/guides/prog_guide/efd_lib.html. (Accessed Jan. 2024).

[21] dtolnay/no-panic GitHub Repository. Accessed May. 2024. RUST no-panic micro. https://github.com/dtolnay/no-panic/tree/master. (Accessed May. 2024).

[22] Gil Einziger and Roy Friedman. 2016. Counting with tinytable: Every bit counts!. In *Proceedings of the 17th International Conference on Distributed Computing and Networking*. 1–10.

[23] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. 2016. Maglev: A fast and reliable software network load balancer. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*. 523–535.

[24] Eric Dumazet. Accessed Jan. 2024. pkt_sched: fq: Fair Queue packet scheduler. https://lwn.net/Articles/564825/. (Accessed Jan. 2024).

[25] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. 2014. Cuckoo Filter: Practically Better Than Bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies (CoNEXT '14)*. Association for Computing Machinery, New York, NY, USA, 75–88. https://doi.org/10.1145/2674005.2674994

[26] Li Fan, Pei Cao, J. Almeida, and A.Z. Broder. 2000. Summary cache: a scalable wide-area Web cache sharing protocol. *IEEE/ACM Transactions on Networking* 8, 3 (2000), 281–293. https://doi.org/10.1109/90.851975

[27] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul Spirakis. 2005. Space efficient hash tables with worst case constant access time. *Theory of Computing Systems* 38, 2 (2005), 229–248.

[28] 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 *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 1069–1084. https://doi.org/10.1145/3314221.3314590

[29] 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 *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021*, James Mickens and Renata Teixeira (Eds.). USENIX Association, 487–501. https://www.usenix.org/conference/nsdi21/presentation/ghigoff

[30] Amélie Gonzalez, Djob Mvondo, and Yérom-David Bromberg. 2023. Takeaways of Implementing a Native Rust UDP Tunneling Network Driver in the Linux Kernel. In *Proceedings of the 12th Workshop on Programming Languages and Operating Systems (PLOS '23)*. Association for Computing Machinery, New York, NY, USA, 18–25. https://doi.org/10.1145/3623759.3624547

[31] Google Cloud. Accessed May. 2024. New GKE Dataplane V2 increases security and visibility for containers. https://cloud.google.com/blog/products/containers-kubernetes/bringing-ebpf-and-cilium-to-google-kubernetes-engine. (Accessed May. 2024).

[32] Pankaj Gupta and Nick McKeown. 2000. Classifying Packets with Hierarchical Intelligent Cuttings. *IEEE Micro* 20, 1 (2000), 34–41. https://doi.org/10.1109/40.820051

[33] 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 *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies, CoNEXT 2018, Heraklion, Greece, December 04-07, 2018*, Xenofontas A. Dimitropoulos, Alberto Dainotti, Laurent Vanbever, and Theophilus Benson (Eds.). ACM, 54–66. https://doi.org/10.1145/3281411.3281443

[34] Nan Hua, Haiquan Zhao, Bill Lin, and Jun Xu. 2008. Rank-indexed hashing: A compact construction of Bloom filters and variants. In *2008 IEEE International Conference on Network Protocols*. 73–82. https://doi.org/10.1109/ICNP.2008.4697026

[35] Qun Huang, Xin Jin, Patrick P. C. Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. 2017. SketchVisor: Robust Network Measurement for Software Packet Processing. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 113–126. https://doi.org/10.1145/3098822.3098831

[36] Intel. Accessed Jan. 2024. Membership Library. https://doc.dpdk.org/guides/prog_guide/member_lib.html. (Accessed Jan. 2024).

[37] Intel. Accessed Jan. 2024. Pktgen - Traffic Generator powered by DPDK. https://github.com/pktgen/Pktgen-DPDK. (Accessed Jan. 2024).

[38] Jinghao Jia, Raj Sahu, Adam Oswald, Dan Williams, Michael V. Le, and Tianyin Xu. 2023. Kernel Extension Verification is Untenable. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HOTOS '23)*. Association for Computing Machinery, New York, NY, USA, 150–157. https://doi.org/10.1145/3593856.3595892

[39] Jonas Otten. Accessed Jan. 2024. Raking the floods: my intern project using eBPF. https://blog.cloudflare.com/building-rakelimit/. (Accessed Jan. 2024).

[40] Kumar Kartikeya Dwivedi. Accessed Jan. 2024. Local kptrs, BPF linked lists. https://lwn.net/Articles/913660/. (Accessed Jan. 2024).

[41] Hsuan-Chi Kuo, Kai-Hsun Chen, Yicheng Lu, Dan Williams, Sibin Mohan, and Tianyin Xu. 2022. Verified programs can party: optimizing kernel extensions via post-verification merging. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 283–299. https://doi.org/10.1145/3492321.3519562

[42] Joshua Levin and Theophilus A. Benson. 2020. ViperProbe: Rethinking Microservice Observability with eBPF. In *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*. 1–8. https://doi.org/10.1109/CloudNet51028.2020.9335808

[43] Jialin Li, Samantha Miller, Danyang Zhuo, Ang Chen, Jon Howell, and Thomas Anderson. 2021. An incremental path towards a safer OS kernel. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 183–190. https://doi.org/10.1145/3458336.3465277

[44] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. 2011. SILT: A Memory-Efficient, High-Performance Key-Value Store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/2043556.2043558

[45] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. 2019. Nitrosketch: Robust and General Sketch-Based Monitoring in Software Switches. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 334–350. https://doi.org/10.1145/3341302.3342076

[46] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 101–114. https://doi.org/10.1145/2934872.2934906

[47] Rodrigo B. Mansilha, Lorenzo Saino, Marinho P. Barcellos, Massimo Gallo, Emilio Leonardi, Diego Perino, and Dario Rossi. 2015. Hierarchical Content Stores in High-Speed ICN Routers: Emulation and Prototype Implementation. In *Proceedings of the 2nd ACM Conference on Information-Centric Networking (ACM-ICN '15)*. Association for Computing Machinery, New York, NY, USA, 59–68. https://doi.org/10.1145/2810156.2810159

[48] 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 (ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 639–653. https://doi.org/10.1145/3620666.3651387

[49] Marek Majkowski. Accessed May. 2024. Cloudflare architecture and how BPF eats the world. https://blog.cloudflare.com/cloudflare-architecture-and-how-bpf-eats-the-world/. (Accessed May. 2024).

[50] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2005. Efficient computation of frequent and top-k elements in data streams. In *Database Theory-ICDT 2005: 10th International Conference, Edinburgh, UK, January 5-7, 2005. Proceedings 10*. Springer, 398–412.

[51] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Mauricio Vásquez Bernal, Yunsong Lu, and Jianwen Pi. 2019. Securing Linux with a faster and scalable iptables. *Comput. Commun. Rev.* 49, 3 (2019), 2–17. https://doi.org/10.1145/3371927.3371929

[52] Sebastiano Miano, Xiaoqi Chen, Ran Ben Basat, and Gianni Antichi. 2023. Fast In-kernel Traffic Sketching in eBPF. *ACM SIGCOMM Computer Communication Review* 53, 1 (2023).

[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. *IEEE Transactions on Network and Service Management* 18, 1 (2021), 133–151. https://doi.org/10.1109/TNSM.2021.3055676

[54] Sebastiano Miano, Alireza Sanaee, Fulvio Risso, Gábor Rétvári, and Gianni Antichi. 2022. Domain Specific Run Time Optimization for Software Data Planes. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 1148–1164. https://doi.org/10.1145/3503222.3507769

[55] Michael Mitzenmacher, Thomas Steinke, and Justin Thaler. 2012. Hierarchical heavy hitters with the space saving algorithm. In *2012 Proceedings of the Fourteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 160–174.

[56] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. 2020. RedLeaf: Isolation and Communication in a Safe Operating System. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 21–39. https://www.usenix.org/conference/osdi20/presentation/narayanan-vikram

[57] Nikita Shirokov, Ranjeeth Dasineni. Accessed May. 2024. Open-sourcing Katran, a scalable network load balancer. https://engineering.fb.com/2018/05/22/open-source/open-sourcing-

katran-a-scalable-network-load-balancer/. (Accessed May. 2024).

[58] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. 2018. Stateless datacenter load-balancing with beamer. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 125–139.

[59] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *Journal of Algorithms* 51, 2 (2004), 122–144. https://doi.org/10.1016/j.jalgor.2003.12.002

[60] 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

[61] Felix Putze, Peter Sanders, and Johannes Singler. 2007. Cache-, hash- and space-efficient bloom filters. In *Experimental Algorithms: 6th International Workshop, WEA 2007, Rome, Italy, June 6-8, 2007. Proceedings 6*. Springer, 108–121.

[62] 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 *Proceedings of the ACM SIGCOMM 2022 Conference (SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 780–794. https://doi.org/10.1145/3544216.3544259

[63] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. 2017. Carousel: Scalable Traffic Shaping at End Hosts. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 404–417. https://doi.org/10.1145/3098822.3098852

[64] Ahmed Saeed, Yimeng Zhao, Nandita Dukkipati, Ellen Zegura, Mostafa Ammar, Khaled Harras, and Amin Vahdat. 2019. Eiffel: Efficient and Flexible Software Packet Scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 17–32. https://www.usenix.org/conference/nsdi19/presentation/saeed

[65] Sanjit Bhat and Hovav Shacham. Accessed Jan. 2024. Formal Verification of the Linux Kernel eBPF Verifier Range Analysis. https://sanjit-bhat.github.io/assets/pdf/ebpf-verifier-range-analysis22.pdf. (Accessed Jan. 2024).

[66] Naveen Kr. Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. 2020. Programmable Calendar Queues for High-speed Packet Scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 685–699. https://www.usenix.org/conference/nsdi20/presentation/sharma

[67] Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. 2003. Packet classification using multidimensional cutting. In *Proceedings of the ACM SIGCOMM 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 25-29, 2003, Karlsruhe, Germany*, Anja Feldmann, Martina Zitterbart, Jon Crowcroft, and David Wetherall (Eds.). ACM, 213–224. https://doi.org/10.1145/863955.863980

[68] V. Srinivasan, S. Suri, and G. Varghese. 1999. Packet Classification Using Tuple Space Search. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '99)*. Association for Computing Machinery, New York, NY, USA, 135–146. https://doi.org/10.1145/316188.316216

[69] The Linux kernel. Accessed Jan. 2024. BPF Kernel Functions (kfuncs). https://docs.kernel.org/bpf/kfuncs.html. (Accessed Jan. 2024).

[70] The Linux kernel. Accessed Jan. 2024. BPF maps. https://docs.kernel.org/bpf/maps.html. (Accessed Jan. 2024).

[71] The Linux kernel. Accessed Jan. 2024. Helper functions. https://www.kernel.org/doc/html/latest/bpf/helpers.html. (Accessed Jan. 2024).

[72] Thomas Gleixner. Accessed Jan. 2024. timer: Refactor the timer wheel. https://lore.kernel.org/lkml/20160617121134.417319325@linutronix.de/. (Accessed Jan. 2024).

[73] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. 2021. revisiting the open vSwitch dataplane ten years later. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 245–257. https://doi.org/10.1145/3452296.3472914

[74] Balajee Vamanan, Gwendolyn Voskuilen, and T. N. Vijaykumar. 2010. EffiCuts: optimizing packet classification for memory and throughput. In *Proceedings of the ACM SIGCOMM 2010 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, New Delhi, India, August 30 -September 3, 2010*, Shivkumar Kalyanaraman, Venkata N. Padmanabhan, K. K. Ramakrishnan, Rajeev Shorey, and Geoffrey M. Voelker (Eds.). ACM, 207–218. https://doi.org/10.1145/1851182.1851208

[75] G. Varghese and T. Lauck. 1987. Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles (SOSP '87)*. Association for Computing Machinery, New York, NY, USA, 25–38. https://doi.org/10.1145/41457.37504

[76] Marcos Augusto M. Vieira, Matheus S. Castanho, Racyus D. G. Pacífico, Elerson Rubens da Silva Santos, Eduardo P. M. Câmara Júnior, and Luiz Filipe M. Vieira. 2021. Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications. *ACM Comput. Surv.* 53, 1 (2021), 16:1–16:36. https://doi.org/10.1145/3371038

[77] 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

[78] Bin Yang, Dian Shen, Junxue Zhang, Fang Dong, Junzhou Luo, and John C.S. Lui. 2022. Towards the Full Extensibility of Multipath TCP with eMPTCP. In *2022 IEEE 30th International Conference on Network Protocols (ICNP)*. 1–11. https://doi.org/10.1109/ICNP55882.2022.9940354

[79] Rui Yang and Marios Kogias. 2023. HEELS: A Host-Enabled eBPF-Based Load Balancing Scheme. In *Proceedings of the 1st Workshop on EBPF and Kernel Extensions (eBPF '23)*. Association for Computing Machinery, New York, NY, USA, 77–83. https://doi.org/10.1145/3609021.3609307

[80] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. 2018. Elastic Sketch: Adaptive and Fast Network-Wide Measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 561–575. https://doi.org/10.1145/3230543.3230544

[81] Tong Yang, Haowei Zhang, Jinyang Li, Junzhi Gong, Steve Uhlig, Shigang Chen, and Xiaoming Li. 2019. HeavyKeeper: An Accurate Algorithm for Finding Top- k Elephant Flows. *IEEE/ACM Transactions on Networking* 27, 5 (2019), 1845–1858. https://doi.org/10.1109/TNET.2019.2933868

[82] Dong Zhou, Bin Fan, Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2013. Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT '13)*. Association for Computing Machinery, New York, NY, USA, 97–108. https://doi.org/10.1145/2535372.2535379

[83] Yang Zhou, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. 2023. Electrode: Accelerating Distributed Protocols with eBPF. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1391–1407. https://www.usenix.org/conference/nsdi23/presentation/zhou